ILU 2.0alpha Reference Manual

Bill Janssen <janssen@parc.xerox.com>
Mike Spreitzer <spreitzer.parc@xerox.com>

(with contributions from Doug Cutting, Frank Halasz, Denis Seversen, and Farrell Wymore)

(typeset 4 January 1996)

This document describes version 2.0alpha of the Inter-Language Unification (ILU) system.

We gratefully acknowledge the contributions of many people, including our reviewers, alpha and beta testers, and regular users. The list includes (but is not limited to): Maria Perez Ayo, Mike Beasley, Erik Bennett, Dan Brotsky, David Brownell, Bruce Cameron, George Carrette, Philip Chou, Daniel W. Connolly, Antony Courtney, Doug Cutting, Paul Everitt, Josef Fink, Steve Freeman, Mark Friedman, Gabriel Sanchez Gutierrez, Jun Hamano, Bruno Haible, Scott W. Hassan, Carl Hauser, Andrew Herbert, Angie Hinrichs, Ben Hurwitz, Roberto Invernici, Swen Johnson, Gabor Karsai, Nobu Katayama, Sangkyun Kim, Ted Kim, Don Kimber, Dan Larner, Carsten Malischewski, Larry Masinter, Fernando D. Mato Mira, Fazal Majid, Steven D. Majewski, Scott Minneman, Masashige Mizuyama, Curtis McKelvey, Chet Murthy, Farshad Nayeri, Les Niles, T. Owen O'Malley, Andreas Paepcke, Jan Pedersen, Karin Petersen, Steve Putz, George Robertson, Joerg Schreck, Ian Smith, Peter Swain, Marvin Theimer, Lindsay Todd, P. B. Tune, Kevin Tyson, Bill van Melle, Guido van Rossum, Brent Welch.

1 ILU Concepts

1.1 What ILU Does

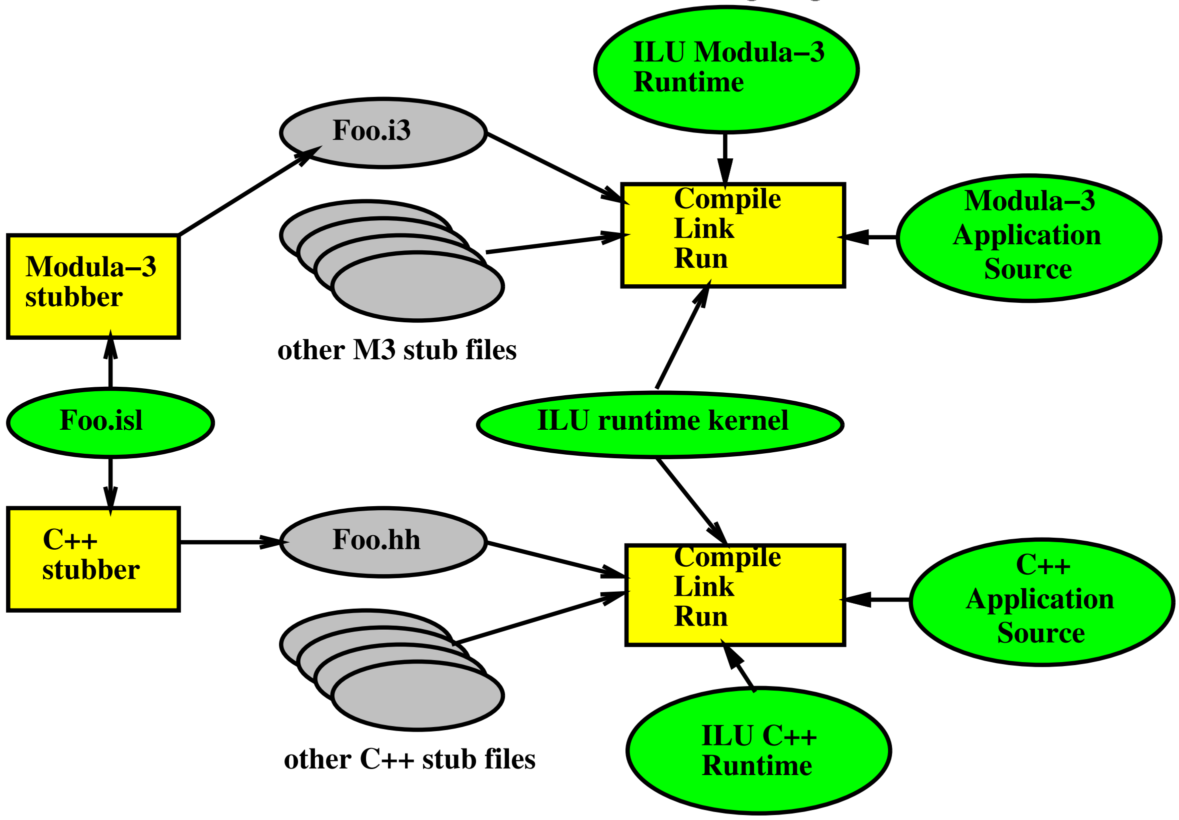
ILU is primarily about interfaces between units of program structure; we call these units *modules*. The notion is that each module enscapsulates some logical part of a program, that has high 'cohesiveness' internally, and low 'coupling' to other parts of the program. ILU provides you with a way of writing down an object-oriented *interface* to the module; that is, the set of object types and other data types that another module would use to communicate with it. This interface can then be processed by various ILU tools to provide customized use of the specified module.

ILU allows many different binding relationships between modules. The modules can be parts of one program instance, all written in the same language; they can be parts written in different languages, sharing runtime support in one memory image; they can be parts running in different program instances on different machines (on different sides of the planet). A module could even be a distributed system implemented by many programs on many machines. A particular module might be part of several different programs at the same time. ILU does all the translating and communicating necessary to use all these kinds of modules in a single program. It optimizes calls across module interfaces to involve only as much mechanism as necessary for the calling and called modules to interact. In particular, when the two modules are in the same memory image and use the same data representations, the calls are direct local procedure calls — no stubs or other RPC mechanisms are involved. The notion of a 'module' should not be confused with the independent concept of a *program instance*; by which we mean the combination of code and data running in one memory image. A UNIX process is (modulo the possibilities introduced by the ability, in some UNIX sytems, to share memory between processes) an example of a program instance.

Because ILU standardizes many of the issues involved in providing proper inter-module independence, such as memory management and error detection and recovery strategies, it can be used to build language-independent *class libraries*, collections of re-usable object definitions and implementations. Because one of the design goals of ILU was to use existing standards for various pieces, rather than inventing anything new, ILU can be used to implement ONC RPC or Xerox Courier services, or clients for existing ONC RPC or Xerox Courier services. ILU also includes an implementation of the Object Management Group's CORBA Internet Inter-Orb Protocol (*IIOP*), and can be used to write CORBA services or clients, as well.

1.2 How ILU Works

The approach used by ILU is one common to standard RPC systems such as Sun's ONC RPC, Xerox's Courier, and most implementations of OMG's CORBA. An interface is described once in some 'language-neutral' interface specification language. Types and exceptions are described; exported functionality is specified by defining *methods* on object types. Tools are then run against the interface description to produce *stubs* for particular programming languages; these stubs can bind to, call, and be called from stubs generated from the same interface description for a different programming language. The stub code is then linked with the application code, some language-specific code containing any necessary ILU support for that programming language, and the *ILU kernel library*, which is code written in ANSI C. The following diagram illustrates the process:

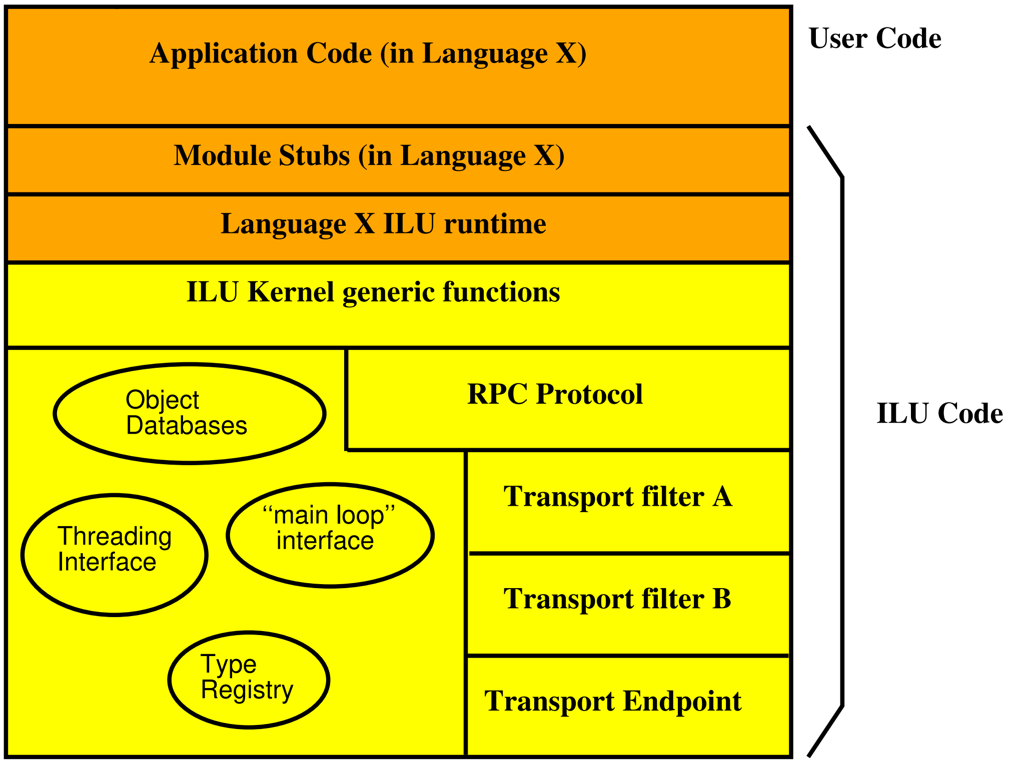


Two ILU modules in different languages

and different address spaces

Several modules may be linked together, for a standalone use. ILU stubs are generated in such a way that applications which link a caller and callee written in the same language directly together suffer no calling overhead. This makes ILU useful for defining interfaces between modules even in programs that do not use RPC.

Different modules of the program may be written in different programming languages. These can either be linked together in the same address space, if the runtimes of the different languages allow that, or they can be used to make separate network servers and clients. In the case of a network service, the memory layout for the program would be something like



An ILU address space

1.3 Core ILU Concepts

1.3.1 Objects

ILU is *object-oriented*. By this, we mean that object types serve as the primary encapsulation mechanism in ILU. All functionality is exported from a module as methods that can be invoked on an instance of some object type, rather than as simple procedures. The object instance provides the context within which methods are executed. The object type system provides subtyping ('inheritance' of interfaces (ILU does not address object implementation)), to aid in structuring of interfaces.

With respect to a particular ILU object instance, a module is called the *server* if it implements the methods of that object, or a *client* if it calls, but does not implement, the methods of that object. One module can thus be a client of one object, and the server of another. An ILU object can be passed as a parameter to or result of a method call, and can be (in) the parameter to an exception. An object may be passed from its server to a client, from a client to its server, or between two clients, in any of the above three kinds of position. Unlike some RPC systems, there can be multiple ILU objects of the same type, even on one machine, even within one program instance.

For a given ILU object, there will, in general, be multiple *language-specific objects*; each is an "object" in one of the programming languages used in the system. One language-specific object, designated the *true object*, actually provides the implementation of the ILU object; it is thus part of the server module. The true object's methods are written by the programmer, not generated by ILU. The other language-specific objects are *surrogate objects*; their methods are actually RPC stubs (generated by ILU) that call on the true object. A surrogate object is used by a client module when the server module is in a different program instance or uses different data representations.

1.3.1.1 Subtyping (interface inheritance)

The object model specified here provides for multiple interface inheritance. It is intended that the subtype provide all the methods described by its supertypes, plus possibly other methods described directly in the subtype description. It is expected that in languages which support multiple-inheritance object models, that an ILU inheritance tree will be reflected in the language-specific inheritance tree. In a single-inheritance language, an ILU-specific multiple-(interface-)inheritance object system must be embedded.

1.3.1.2 Subtype Relationships

In the ILU type system, the only subtyping questions that arise are between two object types. This is because ILU employs only those OOP features common to all languages supported.

Subtyping in ILU is based on structure and name; we include the names in the structure, and thus need only talk about structure. An object type declaration of the form defined later constructs a structure of the form

(OBJTYPE
    SINGLETON: *singleton-protocol-info*
    OPTIONAL: `Boolean`
    COLLECTIBLE: `Boolean`
    AUTHENTICATION: *authentication-type*
    SUPERTYPES: *supertype-structure, ...*
    METHODS: *method-structure, ...*
    LEVEL-BRANDS: (*interface-name*, *interface-brand*,
        type-name, *type-brand*))

Structure A is a subtype of structure B iff either (1) A and B are equal structures, or (2) one member of A's *supertype-structures* is a subtype of B.

Note that the level-brands include the interface name and (optional) brand, as well as the name and (optional) brand of the type being declared. Thus, two declarations of subtypes of the same type normally create distinct subtypes, because they would declare types of different names, or in interfaces with different names. When the interface name and the type name are the same, this does not cause a distinction, although other structural differences might. If the programmer wants to indicate that there's a semantic distinction, even though it doesn't otherwise show up in the structure, s/he can use different interface brands and/or different type brands. These distinctions can be made between declarations in different files, or between successive versions of a declaration in a file that's been edited.

1.3.1.3 Singleton Object Types

Many existing RPC protocols and servers do not have the notion of multiple instances of a type co-existing at the same server, so cannot use the instance discrimination information passed in ILU procedure calls. To support the use of these protocols and servers, we introduce the notion of a *singleton* object type, of which there is only one instance (of each singleton type) at a kernel server. Note that because a single address space may support multiple kernel servers, this means that in a single address space, there may be multiple instances of the same singleton type. When a method is being called on an instance of a singleton type, no instance discrimination information is passed. Singleton types may not be subclassed.

1.3.1.4 Instantiation

To use (e.g., call the methods of) an ILU object, a client must first obtain a language-specific object for that ILU object. This can be done in one of two ways: (1) the client can call on a language-specific object of a different ILU object to return the object in question (or receive the object in a call made on the client, or in the parameter of an exception caught and handled by the client); or (2) certain standard facilities can be used to acquire a language-specific object given either addressing or naming information about the ILU object. The addressing information is called a *string binding handle* (SBH), and the ILU runtime library includes a procedure to acquire a language-specific object given a string binding handle for an ILU object (in strongly-typed languages, this procedure is typed to return an object of the base type common to all ILU objects in that language).

Every creation of a surrogate instance implies communication with the server module, and binding of the surrogate instance to the true instance. ILU may attempt to perform this communication when it is actually necessary, rather than immediately on surrogate instance creation.

The process of creating an instance may bootstrapped via a *name service*, such as the PARC Name-and-Maintenance-Server (`NMS`), which allows servers to register instances on a net-wide basis. A server registers a mapping from naming information to a string binding handle. The client-side stubs for an interface include a procedure that takes naming information, looks up the corresponding string binding handle in the name service, and calls the above-mentioned library routine to map the SBH to a language-specific object. Alternatively, a client can do those steps itself, using an ILU runtime library procedure to acquire a language-specific object for the name service.

1.3.1.5 String Binding Handle

In ILU, there is a string-based representation for a reference to an object. That representation consists of a single string, called a *string binding handle*. ILU uses string binding handles when marshalling object references for RPC. ILU also allows applications to interconvert between objects and string binding handles. This is necessary when dealing with name services, and useful in other circumstances.

A string binding handle contains several different pieces of information:

- The *server ID*, a string which identifies the particular *kernel server* that the object belongs to; any program can separate its objects into one or more groups, each group associated with a different kernel server. Two objects from the same kernel server are called *sibling* objects.
- The *instance handle*, identifies which object on the particular kernel server is being specified.
- The *most specific type ID* (also called the *MSTID*), a type fingerprint for the most specific type of the object.
- The *contact info*, specifies one or more of the ways by which a client of the object can communicate with it.

The server ID, instance handle, and MSTID may each contain any ASCII character other than NUL. They are composed into the string binding handle according the the IETF rules for URLs, but the precise form of the URL is not specified here. (In versions of ILU before 2.0, string binding handles had a completely different syntax.)

The pair (*server ID*, *instance handle*) are also known as the *object ID* (or *OID*) of the object, because together they form a universally unique ID for the object.

The *contact info* part contains one or more contact info sequences, each describing one particular way of communicating with the object's kernel server. Each contact info sequence consists of a series of fields. The first field is known as the *protocol info*, and names a particular RPC protocol, and any parameters that might influence the way in which this protocol would be used. Each of the succeeding fields specifies *transport info*, which defines a way of transforming or communicating data, and any parameters which might influence that transport method. There may be many sequences of contact info in any one string binding handle (but ILU currently ignores all but the first).

1.3.1.6 Siblings

Some ILU object instances may have implementation dependencies on private communication with other instances. For example, imagine an object type `time-share-system`, which provides the method `ListUsers()`, which returns a list of "user" instances. Imagine that `time-share-system` also provides the method `SetUserPriority(u : user, priority : integer)`. We would like to be able to provide some assurance that the user instance used as a parameter to `SetUserPriority` is an instance returned from a call to `ListUsers` on the same instance of a `time-share-system`, because the way in which `SetUserPriority` is implemented relies on the user being a user of that particular `time-share-system`.

The ILU model provides the notion of a *sibling object*. Two instances are siblings if their methods are handled by the same kernel server. Instances that are non-discriminator parameters to methods may be specified in ISL as having to be siblings of the discriminator.

1.3.2 Garbage Collection

A simple form of *garbage collection* is defined for ILU objects. If an object type is tagged as being collectible, a server that implements objects of that type expects clients holding surrogate instances to register with it, passing an instance of a callback object. When a client finishes with the surrogate, the client unregisters itself. Thus the server may maintain a list of clients that hold surrogate instances. If no client is registered for an object, and the object has been dormant (had no methods called on it) for a period of time *T1*, the server may feel free to garbage collect the instance. *T1* is determined by human concerns, not network performance: *T1* is set long enough to allow useful debugging of a client.

To deal with possible failure of a client process, we introduce another time-out parameter. If an instance with registered clients has been dormant for a period of time *T2*, the server uses the callback instance associated with each client to see if the client still exists. If the client cannot be contacted for the callback, the server may remove it from the list of registered clients for that instance.

If a client calls a method on a surrogate instance of a true instance which has been garbage-collected (typically because of partitioning), it will receive the `ilu.ProtocolError` exception, with detail code `ilu.NoSuchInstanceAtServer`.

1.3.3 Connections

ILU, when used to construct distributed systems, has no notion of "connections". That is, the called module has no pointer back to the caller, and no notion of how to do anything with the caller aside from returning a result message. Credentials passed in the request message can identify the caller, but not necessarily the location the call is made from. Protocols that need such information should pass it explicitly as an argument (an instance of a object type with methods defined on it) to the method.

1.3.4 Simple Binding

This release of ILU includes an experimental simple binding/naming facility. It allows a module to *publish* an object, so that another module can import that object knowing only its object ID (as defined in Section 1.4 [ILU Concepts], page 9). The interface to this facility is deliberately quite simple; one reason is to allow various implementations. This release includes one (extremely preliminary) implementation.

The interface consists of three operations: *Publish*, *Withdraw*, and *Lookup*. `Publish` takes one argument, an ILU object. `Publish` returns a string that is needed to successfully invoke `Withdraw`.

`Withdraw` undoes the effects of `Publish`, and takes two arguments: (1) the object in question, and (2) the string returned from `Publish`. In some langauge mappings, the string is not explicitly passed, but conveyed in the language mapping's representation of ILU objects. `Lookup` takes two arguments: an object ID and a type the identified object should have. If the object with that ID is currently being published, and has the given type (among others), `Lookup` returns that object.

The implementation shipped with this release of ILU uses a filesystem directory (specified in the file '*ILUSRC*`/imake/local.defs`') to store information on the currently published objects. To be useful in a distributed system, this directory must be available, by the same name, on all machines in the system.

1.3.5 Error Signalling

ILU uses the notion of an *exception* to signal errors between modules. An exception is a way of passing control outside the normal flow of control. It is typically used for handling of errors. The routine which detects the error signals an exception, which is caught by some error-handling mechanism. The exception type supported in ILU is a termination-model exception, in which the calling stack is unrolled back to the frame which defined the exception handler. Exceptions are signalled and caught using the native exception mechanisms for the servers and clients. A raised exception may carry a single parameter value, which is typed.

1.4 ILU and OMG CORBA

The type and exception model used by ILU is quite similar to that used by the Object Management Group's Common Object Request Broker Architecture (CORBA). We have in fact changed ILU in some ways to more closely match CORBA. Our tools will optionally parse the OMG's Interface Definition Language (OMG IDL) as well as ILU's ISL.

ILU also attempts to address issues that are already upon us, but are not addressed in CORBA 2.0: 64-bit architectures, UNICODE characters, a uniform way of indicating optional values, and garbage collection.

ILU provides two different interface definition languages, OMG IDL and ILU ISL to enhance portability of ILU modules. The OMG IDL subset understood by ILU is a strict subset of OMG IDL; this means that any ILU modules developed using OMG IDL interfaces should be interoperable with any other CORBA system. Any non-CORBA extensions may only be expressed in ILU ISL, so that any modules which use these extensions must use ILU ISL to express their interfaces, thereby underlining the fact that these modules are not CORBA-compliant. We feel that this dual-interface-language approach will tend to enhance both portability and CORBA-compliance of ILU modules.

ILU does not yet provide some of the features required by a full CORBA implementation. Notably it does not provide a Dynamic Invocation Interface, or implementations of either Interface Repository or Implementation Repository. It does not provide the Basic Object Adapter interface, either, but does provide an object adapter with most of the BOA's capabilities, except for those connected with the Interface Repository and/or Implementation Repository.

A number of concepts in CORBA that seem to require further thought are not yet directly supported in ILU: the use of `#include` (ILU uses a more limited notion of "import"); the notion of using an IDL "interface" as both an object type and a name space (this seems to be a "tramp idea" from the language C`++`; in ILU the "interface" defines a name space, and the object type defines a type); the notion that all BOA objects are persistent (in ILU, the question of whether an object is persistent is left up to that object's implementation); the notion that type definitions can exist outside the scope of any module or namespace (in ILU, all definitions occur in some interface). Currently, there is no support in ILU for CORBA `context`s, the OMG IDL type `any`, or the OMG IDL type `Object`. We feel that all three of these notions tend to weaken interface descriptions.

2 Defining Interfaces

Module interfaces may be defined in either the Object Management Group's CORBA OMG IDL, or in ILU's native Interface Specification Language (aka ISL). This document describes the syntax and semantics of ISL, and how to translate OMG IDL interfaces into ISL; see *CORBA 2.0* for a specification of the syntax and semantics of OMG IDL.

2.1 General Syntax of ISL

The conventional file suffix for ISL files is '`.isl`'. Some of the ILU tools rely on the name of the file being the same as the name of the interface defined in it, and rely on having only one interface defined in each '`.isl`' file.

An ISL interface contains four kinds of statements: the interface header, type declarations, exception declarations, and constant declarations. Each statement is terminated with a semi-colon.

Many statements in ISL contain lists: lists of the fields in a record, the types in a union, the methods in an object type. All lists in ISL are terminated with an `END` keyword, and the items in the list are separated by commas.

Comments may be placed in an ISL file. They are introduced with the character sequence `(*`, and terminated with `*)`. Comments nest.

2.1.1 Identifiers

All identifiers that appear in ISL are alphanumeric, begin with an alphabetic character, and may contain hyphens.[1] Differences in case are not sufficient to distinguish between two identifiers; however, the case of an identifier may be preserved in its mapping to a specific programming language.

All ILU type names, exception names, and constant names have two parts, an interface identifier and a local identifier. When writing the full name, the interface identifier comes first, followed by a period, followed by the local identifier. If the interface identifier is ommitted in a name, it defaults to the interface identifier of the most recently encountered interface header.

Interface names, type names, exception names, and constant names occur in different name spaces. Thus is is possible to have a type and an exception with the same name.[2]

---

[1] We might forbid two consecutive hyphens or add other restrictions.

[2] We may change this.

2.1.2 Reserved Words

The following words are reserved words in ISL: `ARRAY`, `ASYNCHRONOUS`, `AUTHENTICATION`, `BOOLEAN`, `BRAND`, `CARDINAL`, `CHARACTER`, `CLASS`, `COLLECTIBLE`, `CONSTANT`, `DEFAULT`, `END`, `ENUMERATION`, `EXCEPTION`, `FALSE`, `FROM`, `FUNCTIONAL`, `IMPORTS`, `IN`, `INOUT`, `INTEGER`, `INTERFACE`, `LIMIT`, `LONG`, `METHODS`, `OBJECT`, `OF`, `OPTIONAL`, `OTHERS`, `OUT`, `RAISES`, `REAL`, `RECORD`, `SEQUENCE`, `SHORT`, `SIBLING`, `SINGLETON`, `SINK`, `SOURCE`, `SUPERCLASS`, `SUPERCLASSES`, `SUPERTYPES`, `TRUE`, `TYPE`, `TYPEID`, `UNION`.

Reserved words may be used as identifiers, by placing them in double quotes, but may not be used as identifiers without quoting.

Other identifiers are worth avoiding, as they may cause problems with specific language implementations. The identifier `t` or `T`, for instance, causes problems with Common Lisp. Language-specific mappings of ISL should try to avoid these problems.

2.2 Statement Syntax

2.2.1 The Interface Header

Each interface is introduced with exactly one interface header of the form

`INTERFACE` *interface-name* [`BRAND` *brand*] [`IMPORTS` *list-of-imported-interfaces* `END`] ;

The *interface-name* is used by various language-specific productions to create name spaces in which the types, exceptions, and constants defined in the interface are declared. The optional *list-of-imported-interfaces* is a comma-separated list of fields, each of the form

interface-name [`FROM` *interface-file*]

where *interface-file* is the typical poorly defined string that names a file for your operating system (in our case, UNIX). Importing an interface allows the current interface to mention the types, exceptions, and constants defined in the imported interface, by referring to them as

interface-name`.`*type-or-value-name*

If the optional "FROM *interface-file*" is not specified for an imported interface, a sensible site-dependent search policy is followed in an attempt to locate that interface, typically looking down a path (environment variable `ILUPATH` on POSIX systems) of directories for a file with the name '*interface-name*`.isl`'.

2.2.2 Type Declarations

In general, a type is defined with a statement of the form

`TYPE` *type-name* `=` *type-reference* `|` *construction* ;

The form `TYPE` *type-name* `=` *type-reference* is used when you want to rename an existing type to make its usage clear or give it a name in the current interface. A *type-reference* is just a *type-name*, or a reference to a type name defined in another interface: *interface-name.type-name*.

2.2.2.1 Primitive types

The following type "names" are pre-defined:

- `INTEGER,` a 32-bit signed integer value;
- `SHORT INTEGER,` a 16-bit signed integer value;
- `LONG INTEGER,` a 64-bit signed integer value;
- `CARDINAL,` a 32-bit unsigned integer value;
- `SHORT CARDINAL`, a 16-bit unsigned integer value;
- `LONG CARDINAL`, a 64-bit unsigned integer value;
- `BYTE`, an unsigned 8-bit byte value;
- `BOOLEAN`, a logical value either True or False;
- `REAL`, an IEEE 64-bit double-precision floating-point value;
- `SHORT REAL`, an IEEE 32-bit single-precision floating-point value;
- `LONG REAL`, a 128-bit quadruple-precision floating-point value;
- `CHARACTER`, a 16-bit UNICODE/IS-10646 character; and
- `SHORT CHARACTER`, an 8-bit ISO 8859-1 character code (but excluding the octet 8_000).

There is also a special type `NULL`, which cannot be used directly; it has a single value, NULL.

2.2.2.2 Constructor overview

The form `TYPE` *type-name* `=` *construction* is used when a user needs to define a new type. Several simple constructors for more complex data types are specified:

- `ARRAY`, a fixed-length N-dimensional array of some specified type;
- `SEQUENCE`, a variable-length one-dimensional array of some specified type;
- `RECORD`, a sequence of typed fields, each of which may be of a different type;
- `UNION`, one of a set of specified types;
- `OPTIONAL`, a union with `NULL`;
- `ENUMERATION`, a type consisting of an explicitly enumerated set of values;
- `OBJECT`, an ILU object type.

In addition, the automatically-imported interface ILU defines the short sequence `CString` of short character.

2.2.2.3 Array Declarations

An `ARRAY` is a fixed-length N-dimensional array of some type. It is defined with a declaration of the form

`TYPE` *type-name* `= ARRAY OF` *dimension-list base-type-reference* `;`

where *dimension-list* is a comma-separated list of non-negative integers, each integer specifying the size of a dimension of the array, and *base-type-reference* is a *type-reference* to some other ILU type. For example,

```
TYPE SymbolTable = ARRAY OF 400 Symbol;
TYPE Matrix3030 = ARRAY OF 30, 30 REAL;
```

The total number of elements in the array may not exceed 4294967295 (2^32-1).

2.2.2.4 Sequence Declarations

A sequence is a variable-length one-dimensional array of some type. It is defined with a declaration of the form

`TYPE` *type-name* `=` [`SHORT`] `SEQUENCE OF` *base-type-reference* [`LIMIT` *size*] `;`

where *base-type-reference* is a *type-reference* to some other ILU type. If the `LIMIT` parameter *size* is used, it limits the sequences to having at most *size* elements; otherwise the sequences are limited to having at most 4294967295 (2^32-1) elements. Use of the `SHORT` modifier is shorthand for a `LIMIT` of 65535 (2^16-1). Use of the `LONG` modifier is not defined for sequences.

2.2.2.5 Generalized Array Declarations

This is a proposed language change, not yet accepted.

The existing language has a weakness: it cannot express coordinated multidimensional variable-length arrays. Coordinated means that there is only one length per dimension, regardless of how many arrays there are at that level. An example is a bitmap of variable height and width: all rows are the same length, and all columns are the same length.

A generalized array type is defined with a declaration of the form

`TYPE` *type-name* `= ARRAY` *dim* , ... *dim* `OF` *base-type-reference* `;`

where each *dim* is of the form

length `|` [`LIMIT` *maxlen* `|` `SHORT`]

A dimension can be given a fixed length by simply specifying that length. A variable-length dimension is either left blank (meaning the maximum length is 2**32-1), specified as `SHORT` (meaning the maximum length is 2**16-1), or given an explicit maximum length.

Note that putting the dimensions after the `OF` would create a syntactic ambiguity in some cases, concerning grouping of a `SHORT`.

2.2.2.6 Record Declarations

`TYPE` *type-name* `= RECORD` *fields...* `END ;`

where *fields* is a comma-separated list of *field*, which has the form

field-name : *field-type-reference*

A sample record declaration:

```
TYPE Symbol = RECORD
  name : string,
  type : TypeInfo,
  address : cardinal
END;
```

2.2.2.7 Union Declarations

A union is a type which may take on values of several different types. To be compliant with the CORBA notion of unions, the union declaration is much more baroque and complicated than it really should be. The declaration has the form:

`TYPE` *type-name* `=` [*tag-type*] `UNION` *arm-list* `END` [`OTHERS`] ;

where *arm-list* is a comma-separated list of *arm*, each of the form:

[*union-case-name* :] *type-name* [*arm-valuator*]

where each *arm-valuator* is either of the form

`= DEFAULT`

or of the form

`=` *value-list* `END`

and where a *value-list* is a comma-separated list of constant values of the tag type. The tag type must be one of: `SHORT INTEGER`, `SHORT CARDINAL`, `INTEGER`, `CARDINAL`, `BYTE`, `BOOLEAN`, or an enumerated type. (We should also allow `SHORT CHARACTER` and `CHARACTER`.) The tag type is `SHORT INTEGER` if not explicitly specified.

A *arm-valuator* must be given for either all or none of the *arms*; if none, the *arms* are assigned single integral values, starting with 0. *arm-valuators* must be given if the tag type isn't numeric. All the values appearing in the *value-lists* of a union must be different from one another. `DEFAULT` can appear in at most one arm of a union type construction. `DEFAULT` and `OTHERS` cannot both appear in the same union.

A union value consists of a tag value, possibly paired with a second value. When the tag value is one that appears in, or is implicitly assigned to, an arm of the union type construction, the second value is of the type named in that arm. Otherwise, the union value is well-formed only if `DEFAULT` or `OTHERS` appears in the union type construction. If an arm is valued with `DEFAULT`, the second value is of that arm's type. If `OTHERS` appears, there is no second value; it is as if there were a default arm of some trivial type (like C's `void` or ML's `unit`).

A simple example:

```
TYPE StringOrInt = UNION ilu.CString, CARDINAL END;
```

A more complex example, that uses an explicit tag type, union case names, and a default arm:

```
TYPE ColorType = ENUMERATION RGB, CMY, HSV, YIQ, HLS END;
TYPE U2 = ColorType
  UNION
    rgb-field : RGBObject = RGB END,
    others : COLORObject = DEFAULT
  END;
```

The union case name is not guaranteed to be present in language-specific mappings.

ISL unions are logically (and sometimes actually, depending on the programming language) tagged. There is a difference between

```
TYPE T1 = UNION Bar, Baz END;
TYPE T2 = UNION Foo, T1 END;
```

and

```
TYPE T1 = UNION Bar, Baz END;
TYPE T2 = UNION Foo, Bar, Baz END;
```

2.2.2.8 Optional Declarations

A variable of type `OPTIONAL Foo` can have either a value of `Foo` or of type `NULL`. It is declared with the form

`TYPE` *type-name* `= OPTIONAL` *base-type-reference* `;`

This should be thought of as roughly equivalent to the declaration

```
TYPE type-name = BOOLEAN UNION base-type-reference = TRUE END END OTHERS ;
```

The difference is that `OPTIONAL` types are logically un-tagged. An optional value is not a pair of (`BOOLEAN`, *base-type-reference*); rather it is a single value, either a special, distinguished, "null" value or a value of the *base-type-reference*. There is thus no difference between

```
TYPE Bar = OPTIONAL Foo;
TYPE Baz = OPTIONAL Bar;
```

and

```
TYPE Bar = OPTIONAL Foo;
TYPE Baz = OPTIONAL Foo;
```

2.2.2.9 Enumeration Declarations

An enumeration is an abstract type whose values are explicitly enumerated. It is declared with the form

```
TYPE type-name = ENUMERATION values... END ;
```

where *values* is a comma-separated list of value names, with optional value ID's that are constants of type `SHORT CARDINAL` that specify the value used to represent the enumeration value "on the wire".[3] *Use of value ID's is deprecated.*

value-name [= *value-id*]

For example,

```
TYPE TapeAction = ENUMERATION
  SkipRecord = 1,
  Rewind = 23,
  Backspace = 49,
  WriteEOF = 0
END;
```

All *value-names* and *value-IDs* must be unique within an enumeration. If *value-IDs* are not assigned explicitly, appropriate values will be assigned automatically in some unspecified way. An enumeration may have at most 65535 (2^16-1) values.

2.2.2.10 Object Type Declarations

Object types are described in the following way:

---

[3] Same integer in all protocols? Yep – for now.

```
TYPE type-name = OBJECT
                 [ SINGLETON protocol-description-string ]
                 [ DOCUMENTATION documentation-string ]
                 [ COLLECTIBLE ]
                 [ OPTIONAL ]
                 [ TYPEID type-id-string ]
                 [ SUPERTYPES supertype-list END ]
                 [ METHODS method-list... END ]
                 [ BRAND string-constant ] ;
```

The keyword `CLASS` is a deprecated synonym for `OBJECT`, and `SUPERCLASSES` is a deprecated synonym for `SUPERTYPES`. Also,

```
[ SUPERCLASS supertype-name ]
```

is a deprecated equivalent to

```
[ SUPERTYPES supertype-name END ]
```

The `SINGLETON` keyword specifies that instances of this type are singleton servers, and implies that the discriminator object (the subject of the call) should not be implicitly marshalled as the first argument in an RPC. This is typically used in describing an instance of an existing RPC service, which is to be modelled in ILU. The argument to `SINGLETON` is a string in the form of ILU "protocol-info", which specifies particular protocol-specific parameters to be used in implementing this object type 'on the wire'. For example, the Sun RPC calendar manager would use a *protocol-description-string* of `"sunrpc_2_100068_3"`, indicating that it uses a Sun RPC program number of 100068 and a Sun RPC version of 3.

The optional *documentation-string* is a quoted string, which is passed on to language-specific bindings where possible, such as with the doc-string capability in Common Lisp.

The `COLLECTIBLE` keyword specifies that instances of this type are meant to be garbage collectible, and that methods necessary for this should be automatically added to its method suite. For an object type to be collectible, all ancestor object types must also be collectible.

The `OPTIONAL` keyword specifies that the language-specific `nil` value may be passed, instead of an instance of this object type, anywhere this object type is used. *This is a CORBA mis-feature, and its use is strongly deprecated. Better to explicitly use a different type constructed with the ILU* `OPTIONAL` *keyword.*

The optional *supertype-list* defines an inheritance relationship between the object types named in the list and the type *type-name*.

The optional *type-id-string* can be used to explicitly assign an MSTID for an object type. Doing so effectively *seals* the object type; that is, changes to the structure of the object type will not be reflected in the MSTID, so version mismatches will not be caught automatically by ILU. This is a dangerous feature (mandated by CORBA).

The *string-constant* in the `BRAND` clause, if any, contributes an arbitrary tag to the structure of the type; omitting the `BRAND` clause is equivalent to giving one with the empty string. Branding gives the programmer a way to make two types distinct despite their otherwise having the same structure. See an earlier subsection for more details.

The *method-list* is a comma-separated list of procedure descriptions. All the methods of an object type have distinct names. This means that independently-developed supertypes might not be usable together.

Methods have the syntax:

> [`FUNCTIONAL`] [`ASYNCHRONOUS`] *method-name* `(` [*args...*] `)`
>   [: *return-type-reference*]
>   [`RAISES` *exceptions...* `END`]
>   [`=` *procedure-id*]
>   [*documentation-string*]

where the discriminator (the implicit first argument to the method, the subject of the call, an instance of the object type in question) is not explicitly listed in the signature. Each method has zero or more arguments in a comma-separated list, each element of which is a colon-separated two-ple

> [*argument-direction*] *argument-name* : [`SIBLING`] *argument-type-reference*

The `SIBLING` keyword may only appear on arguments of an object type, to indicate that the argument should be a sibling object to the discriminator of the method. The `FUNCTIONAL` keyword indicates that the method, for a given set of arguments, is idempotent (i.e., the side effects of one call are the same as the side effects of more than one call) and will always return the same result (or raise the same exception); this information may be used for caching of return values in the client side stubs. The optional *argument-direction* information is one of the three keywords `IN`, `OUT`, `INOUT`, specifying whether the parameter is being used as an input parameter, an output parameter, or both.

A method return type is allowed (again separated from the procedure argument list by a colon), and a list of possible exceptions may be specified as a comma-separated list of exception names, bracketed with the keywords RAISES and END.

The optional *procedure-id* field allows a service description to specify the procedure code that is used in the RPC request packet for this method. Procedure ID's are restricted to the range [0,65279], and must be unique within an interface. This may only be used in methods on objects marked with the `SINGLETON` attribute.

If a method is marked with the `ASYNCHRONOUS` keyword and does not return a value or raise an exception, the RPC method call of a surrogate instance will return after sending the request packet to the RPC partner, as the success of the call does not depend on the completion of the associated code. Other RPC methods will block in such a way as to allow the scheduler to handle other events while it is waiting for the call to complete, if the user has registered the appropriate scheduler hooks with the ILU runtime.

The optional *documentation-string* is a quoted string, which is passed on to language bindings for which it is meaningful, such as the doc-string capability in Common Lisp.

For example:

```
TYPE FancyString = OBJECT
  METHODS
    FUNCTIONAL Length () : cardinal,
    Substring (start : cardinal, end : cardinal) : string
      RAISES StartGreaterThanEnd, StartTooLarge, EndTooLarge END,
    Char (index : cardinal) : character
      RAISES BadIndex END
  END;
```

Note that the object language in ILU is not intended to be used to fully *define* an object type, but rather to *describe* it in a simple language that can be transformed into the different object type definition systems of several other languages.

2.2.3 Exception Declarations

Exceptions in ILU are raised by ILU methods. They allow error conditions to be signalled back to the calling code. They are declared with a statement of the form:

`EXCEPTION` *exception-name* [: *type-reference*] [*documentation-string*] ;

The optional *type-reference* part of the declaration allows the exception to have an associated value, to be used in interpretation of the exception. For example, an exception BadFilename might have the type `ilu.CString`, so that the actual bad filename can be associated with the exception:

The optional *documentation-string* is a quoted string, which is passed on to language bindings for which it is meaningful, such as the doc-string capability in Common Lisp.

```
TYPE Filename = ilu.CString;
EXCEPTION BadFileName : Filename "The value is the bad filename";
```

Because of the uncertain nature of life in distributed systems, the pre-defined exception `ilu.ProtocolError` (defined in the ILU interface) may be raised by any method, to indicate that the method could not be handled, for some reason. It has the following form:

```
TYPE ProtocolErrorDetail = ENUMERATION
                NoSuchClassAtServer = 1,
                BrandMismatch = 2,
                NoSuchMethodOnClass = 3,
                InvalidArguments = 4,
                UnknownObjectInstance = 5,
```

```
                UnreachableModule = 6,
                RequestRejectedByModule = 7,
                TimeoutOnRequest = 8,
                UnknownError = 9
        END;

    EXCEPTION ProtocolError : ProtocolErrorDetail;
```

Signalling of `ProtocolError` is never done by user-written server code; it is reserved to the transport and runtime layers of ILU.

2.2.4 Constant Declarations

For convenience of interface design, constant values for certain simple types may be defined in ISL with statements of the form

`CONSTANT` *constant-name* : *constant-type* `=` *constant-value* ;

2.2.4.1 Integer, Cardinal, and Byte Constants

A *constant-value* for types that are sub-types of `INTEGER`, `CARDINAL`, or `BYTE` is specified with the syntax

[*sign*] [*base-indicator*] *digits*

where the optional *base-indicator* allows selection of bases 2, 8, 10 or 16. It is a digit '0' (zero) followed by either the character 'B' for base 2, 'X' for base 16, 'O' (oh) for base 8, or 'D' for base 10. The *sign* is only valid for subtypes of `INTEGER`; it is either '`+`' or '-'; if not specified, '`+`' is assumed. The *base-indicator* and *digits* fields are case-insensitive.

2.2.4.2 Real Constants

A *constant-value* for subtypes of `REAL` has the syntax:

[*sign*] *integer* . *fraction* [`e` *exponent*]

where *integer* and *fraction* are sequences of decimal digits, *sign* is either '`+`' or '-' ('`+`' is the default), and *exponent* is the power of 10 which the rest of the value is multiplied by (defaults to 0).

2.2.4.3 ilu.CString Constants

A *constant-value* for a sub-type of `ilu.CString` has the form

"*characters*"

where *characters* are any ISO-Latin-1 characters except for 8_000. The escape character is defined to be '#' (hash). The escape character may occur in the string only in the following ways:

#" – a single double-quote character
– a single escape character
#*hex-digithex-digit* – the octet 16_*hex-digithex-digit*
#n – newline
#r – carriage return

2.2.4.4 Examples of Constants

```
CONSTANT Newline : byte = 10;
CONSTANT Pi : short real = 3.14159;
CONSTANT Big : long real = -1.1349e27;  (*  -1.1349 * 10**27   *)
TYPE Filename = ilu.CString;
CONSTANT MyLogin : Filename = "~/.login";
CONSTANT Prompt : ilu.CString = "OK#n ";
CONSTANT HeapBound : cardinal = 0xFFFF39a0;
CONSTANT Pattern1 : cardinal = 0b000001000001;
```

2.3 ilu.isl

The standard interface `ilu` can be found in the file '*ILUHOME*`/interfaces/ilu.isl`'; it is maintained as '*ILUHOME*`/src/stubbers/parser/ilu.isl`'. Here are its contents:

```
INTERFACE ilu BRAND "v1";

TYPE CString = SEQUENCE OF SHORT CHARACTER;

TYPE ProtocolErrorDetail =
  ENUMERATION
    NoSuchClassAtServer,     (* server doesn't handle specified class *)
    BrandMismatch,           (* versions out of sync *)
    NoSuchMethodOnClass,     (* invalid method, or method not implemented *)
    InvalidArguments,        (* bad arguments passed *)
    UnknownObjectInstance,   (* specified instance not on server *)
    UnreachableModule,       (* no path to handler *)
    RequestRejectedByModule, (* request not looked at, for some reason *)
    TimeoutOnRequest,        (* no response from server within timeout *)
    UnknownError             (* catchall error *)
  END;

EXCEPTION ProtocolError : ProtocolErrorDetail;
```

The declarations of `ProtocolErrorDetail` and `ProtocolError` don't belong here, and will be eliminated in favor of a reference manual section explaining the possible errors.

2.4 Using OMG IDL with ILU

ILU allows the use of CORBA OMG IDL[4] instead of ILU ISL. It does this by translating the OMG IDL to its equivalent form in ISL. Most ILU tools will do this automatically, based on whether the suffix of the filename argument is '`.isl`' or '`.idl`'. In addition, the program `idl2isl` can be invoked explicitly to create an ISL version of an OMG IDL interface file. The program idl2isl translates from IDL to ISL. IDL is the Interface Definition Language defined by the Object Management Group.[5]

2.4.1 Translation

On the whole, the translation from IDL to ISL is a straightforward change of syntax. There are a few cases, however, where a bit more is needed.

2.4.1.1 Anonymous types

OMG IDL allows type declarators to be used in certain places in the syntax (for example, struct members and operation parameters). ISL does not; it requires a type name in the corresponding situations. As a result, it is sometimes necessary for the translator to introduce a name in the ISL output for those types that are anonymous in the OMG IDL input. These names are always of the form `AnonType-`*nnn*`-`, where *nnn* is an integer.

For example, the OMG IDL declaration

```
struct str {
long f1;
long f2[5];
};
```

is translated into the following ISL:

```
TYPE AnonType-1- = ARRAY OF 5 INTEGER;
TYPE str = RECORD
f1 : INTEGER,
f2 : AnonType-1-
END;
```

---

[4] OMG IDL is defined in: The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1

[5] The program's front end is derived from the Interface Definition Language Compiler Front End from SunSoft, Inc. See the file '`src/stubbers/idl2isl/Sun-parser/docs/COPYRIGHT`' in the ILU distribution.

2.4.1.2 Topmodules mode

When the translator is in this mode (which it is by default), only `module` declarations are allowed at the topmost level. Each `module` translates into an `INTERFACE` declaration in ISL, and the declarations inside each `module` go into the corresponding ISL `INTERFACE`.

If the translator is not in this mode, all the declarations in the IDL file go into one ISL `INTERFACE` whose name is taken from the OMG IDL input filename, less the '`.idl`' suffix.

2.4.1.3 Imports mode

When the translator is in this mode (which it is by default), `#include` preprocessor directives are, roughly speaking, turned into ISL `IMPORT` statements. This mode allows for separate compilation (stub generation) of interfaces. There are some restrictions: the `#include` directives must occur before any declarations in the file, and the files that are included must not be fragments. That is, each must consist of a sequence of whole declarations (more specifically, `module` declarations if in `topmodules` mode). The included files may in turn include other files.

If the translator is not in this mode, the input is considered to be the result of preprocessing the file first and textually substituting the included files, following the usual behavior of C and C++ compilers.

2.4.1.4 Unsupported constructs

The IDL types `Object` and `any` are disallowed by the translator. Use of `context` clauses on operations is also prohibited.

2.4.2 Manual Invocation of `idl2isl`

The program is run automatically as an intermediate step by any of the ILU tools that take ISL files (normally ending in '`.isl`') if the filename ends in '`.idl`'.

The program may also be run directly, with the following arguments:

`idl2isl { -Wb,`*toggle* `| -Wb,!`*toggle* `}*` *source*`.idl`

In this case, it writes the ISL to its standard output. A toggle is set with an argument `-Wb,`*toggle* and cleared with an argument `-Wb,!`*toggle*. Toggle settings may also be effected by setting the environment variable '`IDL2ISL_OPTS`' to a comma-separated list of toggle names, each of which is either preceded by a '`!`' character (which clears it) or not (which sets it). Command-line arguments take precedence over the environment variable settings.

The toggles are:

- `dump` (default off): produce a dump of the abstract syntax tree. Used for debugging the translator itself.

- `imports` (default on): set the `imports` mode on (explained below).
- `topmodules` (default on): set the `topmodules` mode on (explained below).

2.5 ISL Grammar

In this grammar, parentheses are used for grouping, vertical-bar indicates selection, braces indicated optionality, quotation marks indicate literal keywords or literal punctuation.

No whitespace is allowed between the parts of a `radix`, `number`, or `quoted-string`. Aside from that, whitespace is used to separate fields where necessary, and excess whitespace is ignored outside of `quoted-strings`.

Three primitives are used:

- *name-string*, which is a string consisting of decimal digits, upper and lower-case letters, and hyphens, beginning with a letter. It may not be a keyword, unless it is quoted with double-quotes.
- *string*, which is any sequence of characters.
- *digits*, which is a sequence of digits drawn from the digits for the particular radix. The default radix is decimal.

```
interface = interface-def | interface interface-def

interface-def = interface-declaration other-declarations

interface-declaration = "INTERFACE" name-string
                        [ "BRAND" brand-string ]
                        [ "IMPORTS" import-list "END" ]
                        ";"

import-name = name-string [ "FROM" filename ]

import-list = import-name | import-list "," import-name

other-declarations = constant-decl | exception-decl | type-decl

constant-decl = "CONSTANT" name-string ":" ( integer-const
                                           | cardinal-const
                                           | boolean-const
                                           | byte-const
                                           | float-const
                                           | string-const ) ";"

integer-const = [ "SHORT" | "LONG" ] "INTEGER" "=" [ sign ] number

boolean-const = "BOOLEAN" "=" boolean-value

cardinal-const = [ "SHORT" | "LONG" ] "CARDINAL" "=" number
```

```
byte-const = "BYTE" "=" number

float-const = [ "SHORT" | "LONG" ] "REAL" "="
              [sign] digits [ "." digits ] [ "e" digits ]

number = [ radix ] digits

radix = "0" ( binary | octal | hexadecimal )

binary = "b"

octal = "o"

hexadecimal = "x"

string-const = "ilu.CString" "=" quoted-string

exception-decl = "EXCEPTION" excp-name [ ":" type-name ] [ doc-string ] ";"

excp-name = name-string

type-decl = "TYPE" type-name "=" ( type-name | new-type-decl ) ";"

type-name = primitive-type-name | name-string

primitive-type-name = "BYTE"
                    | [ "SHORT" | "LONG" ] "CARDINAL"
                    | [ "SHORT" | "LONG" ] "INTEGER"
                    | [ "SHORT" | "LONG" ] "REAL"
                    | [ "SHORT" ] "CHARACTER"
                    | "BOOLEAN"

new-type-decl =   record-decl
                | array-decl
                | sequence-decl
                | union-decl
                | optional-decl
                | enum-decl
                | object-decl

record-decl = "RECORD" field-list "END"

field-list = field | field-list "," field

field = name-string ":" type-name

sequence-decl = [ "SHORT" ] "SEQUENCE" "OF" type-name [ "LIMIT" number ]

array-decl = "ARRAY" "OF" dimensions-list type-name
```

```
dimensions-list = number | dimensions-list "," number

union-decl = [ int-enum-or-boolean-type-name ] "UNION" union-field-list "END"

union-field-list = union-field | union-field-list "," union-field

union-field = [ field-name-string ":" ] type-name [ "=" const-list "END" ]

union-field-selector = "=" const-list "END"

const-list = const | const-list "," const

const = number | enum-field-name | boolean-value

enum-decl = "ENUMERATION" enum-field-list

enum-field-list = enum-field | enum-field-list "," enum-field

enum-field = string

boolean-value = "TRUE" | "FALSE"

optional-decl = "OPTIONAL" type-name

object-decl = "OBJECT" object-attributes

object-attributes = object-feature | object-attributes object-feature

object-feature =  "SINGLETON" singleton-protocol-info
                | "COLLECTIBLE"
                | "OPTIONAL"
                | "DOCUMENTATION" doc-string
                | "TYPEID" type-id-string
                | "BRAND" brand-string
                | "SUPERTYPES" supertype-list "END"
                | "METHODS" method-list "END"

supertype-list = type-name | supertype-list "," type-name

singleton-protocol-info = quoted-string

method-list = method | method-list "," method

method = [ "FUNCTIONAL" | "ASYNCHRONOUS" ] name-string
         arguments [ ":" return-type ] [ "RAISES" exception-list "END"]
         [ doc-string ]

return-type = type-name
```

```
exception-list = excp-name | exception-list "," excp-name

arguments = "(" argument-list ")"

argument-list = argument | argument-list "," argument

argument = [ "IN" | "OUT" | "INOUT" ] name-string ":" [ "SIBLING" ] type-name

doc-string = quoted-string

quoted-string = "\"" string "\""
```

3 Using ILU with Common Lisp

This document is for the Common Lisp programmer who wishes to use ILU. The first section explains the mappings from the ILU Interface Specification Language into the Common Lisp language, the second discusses concepts necessary for people exporting ILU modules from Common Lisp. In general, people who need only to use pre-existing ILU modules should only need to read the first section. You should understand the types and concepts supported by ILU before reading this document. (See Section 1.4 [ILU Concepts], page 9.)

3.1 ILU Mappings to Common Lisp

3.1.1 Generating Common Lisp Surrogate and True Stubs

The program ILU `lisp-stubber` takes a interface specification (an '`.isl`' file) and generates lisp code to provide both client-side and server-side support for the interface. The files are generated in the current working directory. In particular, the following files are generated:

- '*interface-name*`-sysdcl.lisp`' – tells `PDEFSYS`[1] how to compile and load the other files. It defines a Common Lisp module `:`<*interface*>, which describes the code needed to support both surrogate and true use of the interface. This file is often called a *sysdcl* for the module.
- '*interface-name*`-basics.lisp`' – contains lisp code needed by clients of the module; and
- '*interface-name*`-server-procs.lisp`' – contains lisp code needed by module implementations.

3.1.2 Packages & Symbols

Runtime code is in the Common Lisp package `ilu`.

Names from interface specifications are transformed into Lisp names (case-insensitive) by inserting hyphens at lower-to-upper case transitions. Hyphens that are already present are maintained as is.[2]

A separate package is defined for each interface with `defpackage`. The name of this package is taken from the name of the interface. This package uses the packages `common-lisp` and `ilu`. The Common Lisp names of all entities defined in the ISL are exported from the package, including types, classes, constants, accessors, type predicates, generic functions, exceptions, etc. Such symbols are also shadowed, to avoid conflicts with used packages. For example, given the following interface:

---

[1] See Section A.1 [The ILU Common Lisp Portable DEFSYSTEM Module], page 132, for a description of the `PDEFSYS` package.

[2] This causes problems; the ISL names "FooBar" and "foo-bar" map to the same Common Lisp name. Something will have to change.

```
INTERFACE MyInterface END;
EXCEPTION TotalLoser : Person;
TYPE Person = OBJECT
  METHODS
     FriendsP (someone : Person) : Boolean
       RAISES TotalLoser END
  END;
```

the stubber generates the following `defpackage`:

```
(defpackage :my-interface
  (:use :common-lisp :ilu)
  (:shadow #:total-loser #:person #:friends-p)
  (:export #:total-loser #:person #:friends-p))
```

This allows symbols defined in the `commonlisp` package to be used by the automatically generated code in the generated package, but it also means that the user needs to be careful about using any generated package. In general, we recommend that you explicitly specify the full name of symbols from ILU interfaces.

3.1.3 Types

ILU types appear in Common Lisp as follows:

1. Exceptions are represented with CL conditions, defined by `define-condition`. All ILU conditions are subtypes of `ilu:rpc-exception`, which is a `serious-condition`. If a value is specified for an exception it may be accessed with `ilu:exception-value` on the condition signalled.
2. Constants are implemented in CL by a value of the appropriate type, defined with `defconstant`.
3. Type aliases (one type name specified as another) are implemented with `deftype`.
4. Sequences are implemented as `lists`, except for sequences of characters, which are implemented as `simple-strings`.
5. BOOLEAN values are represented as normal Common Lisp logical values, typically either `commonlisp:t` or `commonlisp:nil`.
6. Arrays and sequences of CHARACTER (regular or SHORT) are implemented as `simple-strings`.
7. Arrays of bytes are implemented with objects of type `(simple-array (unsigned-byte 8) (*))`.
8. Other arrays are implemented as `simple-arrays`.
9. Unions are implemented as a cons'ed value, with the cdr containing the union type discriminant, and the cdr containing the actual value.
10. Enumerations are implemented with symbols, as in
    `(deftype answer () '(member 'yes 'no 'maybe))`

11. Record types are implemented with CL `defstruct`.
12. Classes are implemented with CLOS `defclass`.

    Private slots are created for methods which are specified as `functional`, and the runtime caches the value of this method in such slots after the first call to the method.

    Methods always take as their first argument the object which they are a method on. Subsequent arguments are those specified in the '`.isl`' file. Methods that have `OUT` or `INOUT` arguments may return multiple values. In general, the parameters to a method are the `IN` and `INOUT` parameters specified in the ISL interface, but not the `OUT` parameters. The return values from a method are the specified return value for the ISL method, if any, followed by the `INOUT` and `OUT` parameters for the method, if any, in the order in which they appear in the ISL specification of the method.

3.2 Using a Module from Common Lisp

To use a module from Common Lisp, you must first have loaded the PDEFSYS "system" that describes the module. Typically, for an ILU interface called *Foo*, the system can be loaded by invoking `(pdefsys:load-system :foo)`. Next, you must bind an instance of an object from that interface. The most common way of doing this is to receive an instance of an object from a method called on another object. But to get the first object exported by that module, one can use either `ilu:sbh->instance` or `ilu:lookup`.

ilu:sbh->instance (*PUTATIVE-TYPE-NAME* `symbol`) (*SBH* `string`) &optional (*MOST-SPECIFIC-TYPE-ID* `simple-string mstid of specified PUTATIVE-TYPE`) => `ilu:rpc-object` Lisp Function

Accepts an ILU string binding handle and Common Lisp type name, and attempts to locally bind an instance of that type with the OID specified in the string binding handle. If no such instance exists locally, a surrogate instance is created and returned. If a true instance exists locally, that instance will be returned.

ilu:lookup (*PUTATIVE-TYPE-NAME* `symbol`) (*OID* `simple-string`) => `(or nil ilu:rpc-object)` Lisp Function

This routine will find and return an object with the OID *OID*, if such an object has been registered in the local domain via the ILU simple binding protocol.[3] See Section 3.7 [Publishing a Common Lisp True Object], page 36, for an example.

Various ILU attributes of an object type may be discovered at run time with the generic function `ilu:ilu-class-info`.

---

[3] The simple binding protocol is experimental in release 1.8 of ILU, and may change without warning in later releases.

ilu:ilu-class-info (*DISC* (or ilu:ilu-object type-name)) (*WHAT* keyword) => (or string boolean list) Lisp Function

This routine will return the specified piece of information about the ILU class specified with *DISC*, which may be either a CLOS class name, or an instance of the class, and with *WHAT*, which identifies which piece of information to return. *WHAT* may have the following values:

- `:authentication` – what kind of authentication, if any, is expected by the methods of this class
- `:brand` – the brand of the object type, if any
- `:collectible-p` – whether or not the object type participates in the ILU distributed GC
- `:doc-string` – the doc string specified for the object type
- `:id` – the ILU unique ID for the object type
- `:ilu-version` – which version of ILU the stubber that generated the code for this object type came from
- `:methods` – a list of the methods of the object type
- `:optional-p` – whether values of this class are allowed to be `cl:nil` (a CORBA excrescence)
- `:name` – the ILU name of the object type

3.3 Implementing a Module in Common Lisp

For each ILU class *interface*.*otype*, ILU will define, in the file '*interface*-server-procs.lisp', a CLOS class called *interface*:*otype*.IMPL. To implement a true object for *interface*.*otype*, one should further subclass this CLOS class, and override all of its methods. In particular, do not let any of the default methods for the class be called from your methods for it.

ILU supports, in each address space, multiple instances of something called a *kernel server*, each of which in turn supports some set of object instances. A kernel server *exports* its objects by making them available to other modules. It may do so via one or more *ports*, which are abstractly a tuple of (*rpc protocol*, *transport type*, *transport address*). For example, a typical port might provide access to a kernel server's objects via `(Sun RPC, TCP/IP, UNIX port 2076)`. Another port on the same kernel server might provide access to the objects via `(Xerox Courier, XNS SPP, XNS port 1394)`.

When creating an instance of a true object, a kernel server for it, and an instance id (the name by which the kernel server knows it) for it must be determined. These may be specified explicitly by use of the keyword arguments to `commonlisp:make-instance` `:ilu-kernel-server` and `:ilu-instance-handle`, respectively. If they are not specified explicitly, the variable `ilu:*default-`

`server*` will be bound, and its value will be used; a default instance handle, unique relative to the kernel server, will be generated.

A kernel server may be created by instantiating the class `ilu:kernel-server`. The keyword argument `:id` may be specified to select a name for the server. Note that ILU object IDs, which consist of the kernel server ID, plus the instance handle of the object on that server, must be unique "across space and time", as the saying goes. If no kernel server id is specified, ILU will generate one automatically, using an algorithm that provides a high probability of uniqueness. If you explicitly specify a kernel server ID, a good technique is to use a prefix or suffix which uniquely identifies some domain in which you can assure the uniqueness of the remaining part of the ID. For example, when using ILU at some project called NIFTY at some internet site in the IP domain `department.company.com`, one might use kernel server IDs with names like `something.NIFTY.department.company.com`.

```
=> (make-instance 'ilu:kernel-server :id "FOO-SERVER-1")
#<ILU:KERNEL-SERVER "FOO-SERVER-1">
=> (make-instance 'ilu:kernel-server)
#<ILU:KERNEL-SERVER "121.2.100.231.1404.2c7577eb.3e5a28f">
=>
```

ilu:kernel-server &key (*id* `string` `nil`) (*unix-port* `fixnum` `0`) (*object-table* `list of 2 elements` `nil`) => `ilu:kernel-server` — Lisp `cl:make-instance`

Creates and returns an instance of `ilu:kernel-server`. If *id* is specified, the server has that value for its server ID. If *unix-port* is specified, the server attempts to 'listen' on that UNIX port, if the notion of a UNIX port is applicable. If *object-table* is specified, it must consist of a list of two functions. The first function must take a string, which is the instance handle of a desired object on this kernel server, and return a value of type `ilu:ilu-true-object`. The second funtion must free up any resources used by this object table.

To export a module for use by other modules, simply instantiate one or more instances of your subtypes of *interface*:*otype*.`IMPL` (which will inherit from `ilu:ilu-true-object`.

```
=> (make-instance 'foo:my-bar.impl :ilu-kernel-server s)
#<FOO:MY-BAR.IMPL 0x3b32e8 "1">
=>
```

ilu:ilu-true-object &key (*ilu-kernel-server* `ilu:kernel-server` `nil`) (*ilu-instance-handle* `string` `nil`) => `ilu:ilu-true-object` — Lisp `cl:make-instance`

Creates and returns an instance of `ilu:ilu-true-object`. If *ilu-true-server* is specified, the instance is created on the specified server. If *ilu-instance-handle* is specified, that instance handle is used.

The simplest Common Lisp "server" code would look something like:

```
(defun start-server ()
  (make-instance 'foo:my-bar.impl))
```

which will create an instance of `FOO:MY-BAR.IMPL` and export it via a default server.

It's also possible to find out who is making the call:

ilu:*caller-identity* Lisp Variable

The identity of the caller is bound to the special variable `ilu:*caller-identity*`. It is a string which begins with the name of an identity scheme, followed by an identity in that scheme. For example, an identity in the SunRPC UNIX identity scheme would be something like `"sunrpc-unix:2345,67@13.12.11.10"` (i.e., `"sunrpc-unix:`<*uid*>`,`<*gid*>`@`<*hostname*>`"`). If no identity is furnished, a zero-length string is bound.

3.3.1 Publishing

To enable users of your module find the exported objects, you may register the string binding handle of the object or objects, along with their type IDs, in any name service or registry that is convenient for you. In release 1.6 of ILU, we are supporting an experimental simple binding method that allows you to "publish" an object, which registers it in a domain-wide registry, and then to withdraw the object, if necessary. Potential clients can find the string binding handle and type ID of the object by calling a lookup function. **Note that this interface and service is experimental, and may be supported differently in future releases of the ILU system.**

ilu:publish (*OBJ* `ilu:rpc-object`) => `boolean` Lisp Function

Accepts an `ilu:rpc-object` instance and registers it with some domain-wide registration service. The object is known by its

object ID (OID), which is composed of the ID of its kernel server, plus a server-relative instance ID, typically composed as *instance-ID*@*server-ID*. Clients may find the object by looking up the OID via the `ilu:lookup` function. The function returns non-`cl:nil` if the publication succeeded.

ilu:withdraw (*OBJ* `ilu:rpc-object`) => `boolean` Lisp Function

If *OBJ* is registered, and if it was published by the same address space that is calling `withdraw`, its registration is withdrawn. The function returns non-`cl:nil` if the object is no longer published.

If you wanted to create an instance, and publish it, the code for starting a service might look something like this:

```
(defun start-server ()
  (let* ((ks (make-instance 'ilu:kernel-server
```

```
                ;; specify the service id
                :id "service.localdomain.company.com"))
         (si (make-instance 'foo:my-bar.impl
                ;; specify the server
                :ilu-kernel-server ks
                ;; specify the instance handle
                :ilu-instance-handle "theServer")))
    ;; the OID for "si" is now "theServer@service.localdomain.company.com"
    (ilu:publish si)
    si))
```

Someone who wanted to use this service could then find it with the following:

```
(defun find-server ()
  (ilu:lookup 'foo:bar "theServer@service.localdomain.company.com"))
```

3.3.2 Debugging

To help with finding errors in your methods, the variable `*debug-uncaught-conditions*` is provided.

ilu:*debug-uncaught-conditions* Variable

If `cl:t`, causes a server to invoke the debugger when an unhandled error in user code is encountered, rather than the default action of signalling an exception back to the caller. The default value is `cl:nil`.

3.4 Dumping an image with ILU

ILU has dynamic runtime state. In particular, after it is initialized, it uses several Common Lisp threads to maintain part of its state, and may also keep open connections on operating system communication interfaces. If you wish to dump an image containing ILU, you must dump the image before initializing the `ILU` module.

Initialization occurs automatically whenever a instance of `ilu:ilu-object` or `ilu:rpc-server` is created. Thus you should not create any instances of either true or surrogate ILU objects before dumping the image. However, you may load all the interface code for any interfaces that you are using, before dumping the image.

Initialization may also be accomplished by an explicit call to `ilu:initialize-ilu`:

ilu:initialize-ilu Lisp Function

Initializes the `ILU` module.

You may check to see whether the system has been initialized by examining the variable `ilu::*ilu-initialized*`, which is `t` iff `ilu:initialize-ilu` has been invoked.

3.5 The Portable DEFSYSTEM Module

ILU support uses a portable implementation of `DEFSYSTEM` to specify modules to Common Lisp. See Section A.1 [The ILU Common Lisp Portable DEFSYSTEM Module], page 132, for details of this system.

3.6 ILU Common Lisp Lightweight Processes

ILU currently assumes the existence of lightweight process, or thread, support in your Common Lisp implementation. It uses these internally via a generic veneer, described fully in Section B.11 [The ILU Common Lisp Lightweight Process System], page 152.

3.7 Porting ILU to a New Common Lisp Implementation

The Lisp support provided with ILU includes support for the Franz Allegro Common Lisp 4.x implementation. To use ILU with other Common Lisp implementations, please see Section C.5 [Porting ILU to Common Lisp Implementations], page 158.

4 Using ILU with C++

4.1 Introduction

This document is for the C++ programmer who wishes to use ILU. The following sections will show how ILU is mapped into C++ constructs and how both C++ clients and servers are generated and built.

When functions are described in this section, they are sometimes accompanied by *locking comments*, which describe the locking invariants maintained by ILU on a threaded system. See the file '*ILUSRC*`/runtime/kernel/iluxport.h`' for more information on this locking scheme, and the types of locking comments used.

A number of macros are used in function descriptions, to indicated optional arguments, and ownership of potentially malloc'ed objects. The owner is responsible for freeing the object's storage at an appropriate time, and generally makes no interesting guarantees of when that will be. Some types of objects (generally (fixed- and variable-length) arrays, including strings) are presumed to be mutable by only their owners. The macro `OPTIONAL(`*type-name*`)` means that the value is either of the type indicated by *type-name*, or the value `NULL`. This macro may only be used with pointer values. The macro `RETAIN(`*type-name*`)` indicates, when used on a parameter type, that the caller retains ownership of the value, and when used on a return type, that the called function retains ownership of the value. The macro `PASS(`*type-name*`)` indicates, when used on a parameter type, that the caller is passing ownership of the storage to the called function, and when used on a return type, that the called function is passing ownership of the called value to the caller. The macro `GLOBAL(`*type-name*`)` means that neither the caller nor the calling function owns the storage.

4.2 Mapping ILU ISL to C++

Using ILU with C++ is intended to eventually be compatible with the OMG CORBA specification. That is, all of the naming and stub generation comply with the Common Object Request Broker Architecture specified mapping for C++, when that specification is available. The current mapping was designed to be usable with a large number of C++ compilers, by avoiding problematic constructs such as templates, exceptions, namespaces, and nested class definitions.

Note that ILU support for C++ does rely on having argument prototypes, all C++ library functions, and the capabilities of the C++ pre-processor.

4.2.1 Names

In general, ILU constructs C++ names from ISL names by replacing hyphens with underscores. Type names are prepended with their interface name and the string "_T_". Enumeration value namess are formed by prepending the enumeration type name and "_" to the ISL enumeration value name. Exception names are prepended with their interface name and "_E_". Constant names are prepended with their interface name and "_C_".

Other naming conventions may be specified explicitly; see the following section on tailoring names for more information.

4.2.2 Types

Records turn directly into C++ structs. Unions consist of a struct with two fields: the type discriminator, a field called "discriminator", and a union of the possible values, called "value". Arrays map directly into C++ arrays. Sequences become a C++ class with methods and representation analogous to the procedures and representation that appear in the C mapping. Objects become normal C++ classes that are subclasses of the pre-defined class `iluObject`.

4.2.2.1 Sequence types

For most sequences types, the generated C++ code follows a pattern analogous to that set in the C mapping and illustrated in the upcoming example.

4.2.3 Object types

4.2.4 Exceptions

Because of the scarcity of implementation of the C++ exception mechanism, exceptions are passed by adding an additional argument to the beginning of each method, which is a pointer to a status struct, which contains an exception code, and a union of all the possible exception value types defined in the interface. Method implementations set the exception code, and fill in the appropriate value of the union, to signal an exception. Exception codes are represented in C++ with values of the type `ilu_Exception`.

In a true module, exceptions may be raised by using the function `<`*interface*`>_G::RaiseException`.

`void` `<`*interface*`>`**_G::RaiseException** (`RETAIN(<`*interface*`>Status *)` *status*, `GLOBAL(ilu_Exception)` *code*, `...`) C++

Causes an exception code and value for the exception specified by *code* to be bound in *status*. Besides the two required arguments, the function may take another argument, which should be a value of the type implied by the value of *code*; that is, of the appropriate type to be a value of the exception being signalled. Note that `RaiseException` does not actually cause a transfer of control, so that an explicit return statement must follow a call to `RaiseException`.

4.2.5 Constants

Constants are implemented with C++ `#define` statements.

4.2.6 Examples

Here's a sample ISL spec, and the resulting C++ mappings:

```
INTERFACE Foo;

TYPE String = ilu.CString;
TYPE UnsignedInt = CARDINAL;

TYPE E1 = ENUMERATION val1, val2, val3=40 END;
TYPE R1 = RECORD field1 : CARDINAL, field2 : E1 END;
TYPE A1 = ARRAY OF 200 BYTE;
TYPE A2 = ARRAY OF 41, 3 R1;
TYPE S1 = SEQUENCE OF E1;
TYPE U1 = UNION R1, A2 END;

EXCEPTION Except1 : String;

CONSTANT Zero : CARDINAL = 0;

TYPE O1 = OBJECT
  METHODS
    M1 (arg1 : R1) : UnsignedInt RAISES Except1 END
  END;
```

The C++ mapping:

```
typedef ilu_CString Foo_T_String;
typedef ilu_Cardinal Foo_T_UnsignedInt;

typedef enum _Foo_T_E1_enum {
  Foo_T_E1_val1 = 1,
  Foo_T_E1_val2 = 2,
  Foo_T_E1_val3 = 40
} Foo_T_E1;
typedef struct _Foo_T_R1_struct {
  ilu_Cardinal field1;
  Foo_T_E1 field2;
} Foo_T_R1;
typedef ilu_Byte Foo_T_A1[200];
typedef Foo_T_R1 Foo_T_A2[41][3];
class _Foo_T_S1_sequence {
 private:
  ilu_Cardinal _maximum;
  ilu_Cardinal _length;
  Foo_T_E1 *_buffer;
 public:
  _Foo_T_S1_sequence ();
  virtual ~_Foo_T_S1_sequence ();
  static class _Foo_T_S1_sequence *Create (ilu_Cardinal initial_size, Foo_T_E1
*initial_data);
```

```
  virtual ilu_Cardinal Length();
  virtual void Append(Foo_T_E1);
  virtual Foo_T_E1 RemoveHead();
  virtual Foo_T_E1 RemoveTail();
  virtual ilu_Cardinal RemoveAll(ilu_Boolean (*matchproc)(Foo_T_E1));
  virtual Foo_T_E1 * Array();
  virtual Foo_T_E1 Nth(ilu_Cardinal index);
};
typedef class _Foo_T_S1_sequence * Foo_T_S1;
enum Foo_T_U1_allowableTypes { Foo_T_U1_R1, Foo_T_U1_A2 };
typedef struct _Foo_T_U1_union {
  enum Foo_T_U1_allowableTypes discriminator;
  union {
    Foo_T_R1 R1;
    Foo_T_A2 A2;
  } value;
} Foo_T_U1;

extern ilu_Exception Foo_E_Except1;     /* exception code Except1 */

typedef struct _Foo_Status_struct {
  ilu_Exception returnCode;
  union {
    ilu_Cardinal anyvalue;
    Foo_T_String Except1;
  } values;
} FooStatus;

class Foo_T_O1 : public iluObject {
 public:
  Foo_T_O1(); // constructor
  virtual ~Foo_T_O1(); // destructor
  static class Foo_T_O1 * ILUCreateFromSBH(ilu_CString sbh);
  Foo_T_UnsignedInt M1 (FooStatus *_status, Foo_T_R1 *arg1);
};

#define Foo_C_Zero ((ilu_Cardinal) 0)
```

4.3 Using an ILU module from C++

A client module may obtain an instance of an ILU object in three basic ways: 1) instantiating it directly from a string binding handle, 2) using the function `iluObject::Lookup` to locate it via the simple binding interface, and 3) receiving the instance directly as a return value or out parameter from a method on a different object.

To instantiate from a string binding handle, a static member function is generated for each subclass of `class iluObject` declared in the C++ stubs:

`OPTIONAL(class` *T* `*)` **iluObject::ILUCreateFromSBH** `(ilu_CString` *sbh*`)` C++

To use the simple binding service to locate an object:

`static OPTIONAL(GLOBAL(void *))` **iluObject::Lookup** `( RETAIN(char *)` *sid*`, RETAIN(char *)` *ih*`, ilu_Class` *putative-class* `)` C++

Locking: Main invariant holds.

Finds and returns the object specified by the given Server ID (*sid*) and server-relative Instance Handle (*ih*) by consulting the local domain registry of objects. *putative-class* is the type that the object is expected to be of, though the type of the actual object returned may be a subtype of *putative-class*, cast to the *putative-class*. The return value should be immediately cast to a value of the C++ mapping of *putative-class*.

4.4 Implementing an ILU Module in C++

For each ILU class *interface*`.`*otype*, ILU will define, in the file '*interface*`.cc`', a C++ class called *interface*`_T_`*otype*. To implement a true object for *interface*`.`*otype*, one should further subclass this C++ class, and override all of its methods. In particular, do not let any of the default methods for the class be called from your methods for it.

4.4.1 Servers

ILU supports, in each address space, multiple instances of something called a *kernel server*, each of which in turn supports some set of object instances. A kernel server *exports* its objects by making them available to other modules. It may do so via one or more *ports*, which are abstractly a tuple of (*rpc protocol*, *transport type*, *transport address*). For example, a typical port might provide access to a kernel server's objects via `(Sun RPC, TCP/IP, (host 13.24.52.9, UNIX port 2076))`. Another port on the same kernel server might provide access to the objects via `(Xerox Courier, XNS SPP, XNS port 1394)`.

When creating an instance of a true object, a kernel server for it, and an instance handle (the name by which the kernel server knows it) for it must be determined. These may be specified explicitly by overriding the default `iluObject::ILUGetServer` and `iluObject::ILUGetInstanceHandle` methods, respectively. The `iluObject` implementation of `ILUGetServer` defers to `ilu::GetDefaultServer`. The `iluObject` implementation of `ILUGetInstanceHandle` generates a handle that's unique relative to the kernel server.

The kernel server is represented in C++ with the class `iluServer`, which has the following constructor:

?? **iluServer::iluServer** (`OPTIONAL(const char *)` *server-id*, `OPTIONAL(iluObjectTable *)` *object-table*) C++

Constructs an instance of `class iluObject` with the given *server-id* and *object-table*.

Note that ILU object IDs, which consist of the kernel server ID, plus the instance handle of the object on that server, must be unique "across space and time", as the saying goes. If no kernel server id is specified, ILU will generate one automatically, using an algorithm that provides a high probability of uniqueness. If you explicitly specify a kernel server ID, a good technique is to use a prefix or suffix which uniquely identifies some domain in which you can assure the uniqueness of the remaining part of the ID. For example, when using ILU at some project called NIFTY at some internet site in the IP domain `department.company.com`, one might use kernel server IDs with names like `something.NIFTY.department.company.com`.

Once the server is constructed, a port must be added:

`ilu_Boolean` **iluServer::AddPort** (`OPTIONAL(RETAIN(char *))` *protocol-info*, `OPTIONAL(RETAIN(ilu_TransportInfo))` *transport-info*, `ilu_Boolean` *be-default*) C++

Adds a port through which the server can be contacted. The *protocol-info* and *transport-info* specify the RPC and transport protocols and their parameters. The *transport-info* has a layered structure, represented by the C type `ilu_TransportInfo`, described in '*ILUSRC*`/runtime/kernel/iluxport.h`'. See chapter 1 for a catalogue of available RPC and transport layer specifications.

To export a module for use by other modules, simply instantiate one or more instances of your subtype of *interface*:*otype* and (if single-threaded) call the ILU C++ event dispatching loop, `iluServer::Run`.

4.4.2 Event dispatching

Most non-threaded long-lived C and C++ programs simulate threads with *event dispatching*, in which the program waits in some piece of code called the *main loop* until an *event* such as input arriving on a file descriptor or the expiration of an alarm signal causes a *callback routine* to be invoked. The ILU C++ runtime, in single-threaded mode, supports this style of operation with various static member functions of the class `iluServer`.

`static ilu_Boolean` **iluServer::RegisterInputHandler** (`int` *fd*, `void (*`*callbackRoutine*`)(int, void *)`, `void *` *callbackArg*) C++

Register the file descriptor *fd* with the ILU kernel so that when ILU kernel event dispatching is active (that is, during the `iluServer::Run` call), the function *callbackRoutine* will be invoked with the arguments (*fd*, *callbackArg*) whenever input is available on the file descriptor *fd*.

`static ilu_Boolean` **iluServer::UnregisterInputHandler** (`int` *fd*) C++
: Removes any callback routine registered on file descriptor *fd*.

`static ilu_Boolean` **iluServer::Run** (`void`) C++
: Invokes the ILU main loop and causes ILU kernel event dispatching to be active. This routine never returns.

Occasionally it is necessary to use a different event dispatching mechanism, typically because some other work is done inside the main loop of the mechanism. An alternate main loop can be registered for use with ILU by creating a subtype of the class `iluMainLoop` and registering it with the kernel by calling the function `iluServer::iluSetMainLoop`:

`static void` **iluServer::iluSetMainLoop** (`RETAIN(iluMainLoop *)` *ml*) C++
: Registers the main loop object *ml* with the runtime kernel.

4.4.3 Publishing

To enable users of your module find the exported objects, you may register the string binding handle of the object or objects, along with their type IDs, in any name service or registry that is convenient for you. In releases 1.6–2.0 of ILU, we support an experimental simple binding method that allows you to "publish" an object, which registers it in a domain-wide registry, and then to withdraw the object, if necessary. Potential clients can find the string binding handle of the object by calling a lookup function. **Note that this interface and service is experimental, and may be supported differently in future releases of the ILU system.**

`ilu_Boolean` **iluObject::ILUPublish** () C++
: A method on instances of `class iluObject`, it registers the instance with some domain-wide registration service. The object is known by its <*Server-ID*, *Instance-Handle*> pair. Clients may find the object by passing this pair to the `iluObject::Lookup` function. Returns true if the object can be successfully published in the local registry.

`ilu_Boolean` **iluObject::ILUWithdraw** () C++
: Returns true if the object's registration in the local registry can be successfully withdrawn, or does not exist.

4.5 ILU API for C++

4.6 Generating ILU stubs for C++

To generate C++ stubs from an ISL file, you use the program `c++-stubber`. Three files are generated from the '`.isl`' file (the extension `cpp` is used instead of `cc` when running on Windows):

- '*interface-name*`.hh`' contains the class definitions for the types and procedures defined by the interface;
- '*interface-name*`.cc`' contains the client-side and general code for the interface; and
- '*interface-name*`-server-stubs.cc`' contains the server-side stubs and code for the interface.

Typically, clients of a module never have a need for the '*interface-name*`-server-stubs.cc`' file.

```
% c++-stubber foo.isl
header file interface foo to ./foo.hh...
code for interface foo to ./foo.cc...
code for server stubs of interface foo to ./foo-server-stubs.cc...
%
```

The option `-renames` *renames-filename* may be used with `c++-stubber` to specify particular C++ names for ISL types. See the following section for more details.

4.6.1 Tailoring C++ Names

It is sometimes necessary to have the C++ names of an ILU interface match some other naming scheme. A mechanism is provided to allow the programmer to specify the names of C++ language artifacts directly, and thus override the automatic ISL to C++ name mappings.

To do this, you place a set of synonyms for ISL names in a

renames-file, and invoke the `c++-stubber` program with the switch `-renames`, specifying the name of the renames-file. The lines in the file are of the form

construct ISL-name C++-name

where *construct* is one of `method`, `exception`, `type`, `interface`, or `constant`; *ISL-name* is the name of the construct, expressed either as the simple name, for interface names, the concatenation *interface-name*`.`*construct-name* for exceptions, types, and constants, or *interface-name*`.`*type-name*`.`*method-name* for methods; and *C++-name* is the name the construct should have in the generated C++ code. For example:

```
# change "Foo_T_R1" to plain "R1"
type Foo.R1 R1
# change name of method "M1" to "Method1"
method Foo.O1.M1 Method1
```

Lines beginning with the 'hash' character '`#`' are treated as comment lines, and ignored, in the renames-file.

This feature of the `c++-stubber` should be used as little and as carefully as possible, as it can cause confusion for readers of the ISL interface, in trying to follow the C++ code. It can also create name conflicts between different modules, unless names are carefully chosen.

4.7 Threading

The ILU C++ runtime is prepared to be used in either a single-thread or a multi-threaded mode. Single-threaded is the default. To run multi-threaded, the application is responsible for picking a threading mechanism and making it accessible to ILU. Two calls must be made to make a threading mechanism available to ILU. One, `iluServer::SetFork`, enables ILU to fork new threads. The other, `ilu_SetLockTech` (from '*ILUSRC*`/runtime/kernel/iluxport.h`'), supplies various thread synchronization primitives. These calls must be done at startup time, before any ILU servers are created or surrogate objects imported.

4.8 Other ILU Considerations For C++

4.8.1 Libraries and Linking

For clients of an ILU module, it is only necessary to link with the '*interface-name*`.o`' file compiled from the '*interface-name*`.cc`' file generated for the interface or interfaces being used, and with the two libraries '*ILUHOME*`/lib/libilu-c++.a`' and '*ILUHOME*`/lib/libilu.a`' (in this order, as '`libilu-c++.a`' uses functions in '`libilu.a`').

For implementors of true classes, or servers, the code for the server-side stubs, in the file '*interface-name*`-server-stubs.o`', compiled from '*interface-name*`-server-stubs.cc`', should be included along with the other files and libraries.

4.8.2 Initialization order

ILU uses the static-object-with-constructor trick to effect per-compilation-unit startup code. In certain cases you'll want to ensure that a certain compilation unit's initialization is run before another's. While C++ defines no standard way to do this, most compilers work like this: compilation units are initialized (static object construtors run) in the order in which they are given to the link-editor. We want to hear about any exceptions to this rule.

4.8.3 Makefiles

ILU uses the `imake` system from X11R? to produce '`Makefile`'s from '`Imakefile`'s. For more details on this process, Section 14.4.1 [Using Imake with ILU], page 124.

5 Using ILU with ANSI C

5.1 Introduction

This document is for the ANSI C programmer who wishes to use ILU. The following sections will show how ILU is mapped into ANSI C constructs and how both ANSI C clients and servers are generated and built.

Using ILU with ANSI C is intended to be compatible with the OMG CORBA specification. That is, all of the naming and stub generation comply with the Common Object Request Broker Architecture, revision 2.0.[1]

Note that ILU does not support non-ANSI variants of the C language. In particular, it relies on having prototypes, all ANSI C library functions, and the capabilities of the ANSI C pre-processor.

When functions are described in this section, they are sometimes accompanied by *locking comments*, which describe the locking invariants maintained by ILU on a threaded system. See the file '*ILUHOME*`/include/iluxport.h`' for more information on this locking scheme, and the types of locking comments used.

A number of macros are used in function descriptions, to indicated optional arguments, and ownership of potentially malloc'ed objects. The macro `OPTIONAL(`*type*`)` means that the value is either of the type indicated by *type*, or the value `NULL`. This macro may only be used with pointer types. The macro `RETAIN(`*type*`)` indicates, when used on a parameter, that the caller retains ownership of the value, and when used in the result position, that the called function retains ownership of the value. The macro `PASS(`*type*`)` indicates, when used on a parameter, that the caller is passing ownership of the storage to the called function, and when used in the result position, that the called function is passing ownership of the called value to the caller. The macro `GLOBAL(`*type*`)` means that neither the caller nor the calling function owns the storage.

[1] The Common Object Request Broker: Architecture and Specification, OMG document number 93.12.43 (is this still right??), revision 2.0, Draft July 1995

5.2 The ISL Mapping to ANSI C

5.2.1 Names

In general, ILU constructs ANSI C names from ISL names by replacing hyphens with underscores. Type names and class names are prepended with their interface name. For example, for the ISL type `T-1` in interface `I`, the generated name of the ANSI C type would be `I_T_1`.

Enumeration value names are formed by prepending the interface name and "_" to the ISL enumeration value name. Enumeration names and values are then cast into ANSI C `enum` statements.

Constant names are prepended with their interface name. They are implemented with the `const` declaration statements.

Method name prefixes are specified by CORBA to be *module-name_interface-name*. ANSI C function names for ISL methods are composed of the generated class name prepended to the method name. For example, if the interface name is `X` and the class type name is `Y` and the ISL method name is `Z` then the ANSI C callable method name will be `X_Y_Z`. ILU ANSI C servers for this method must implement a function called `server_X_Y_Z`.

For field names within records, hyphens are replaced with underscores.

5.2.2 Mapping Type Constructs Into ANSI C

5.2.2.1 Basic Types

The following basic ISL types have the corresponding mappings in ANSI C, as specified by the CORBA 2.0 standard mapping for C:

- `BOOLEAN` => `CORBA_boolean`
- `BYTE` => `CORBA_octet`
- `CHARACTER` => `ilu_character`
- `SHORT CHARACTER` => `CORBA_char`
- `CARDINAL` => `CORBA_unsigned_long`
- `SHORT CARDINAL` => `CORBA_unsigned_short`
- `LONG CARDINAL` => `ilu_longcardinal`
- `INTEGER` => `CORBA_long`
- `SHORT INTEGER` => `CORBA_short`
- `LONG INTEGER` => `ilu_longinteger`
- `REAL` => `CORBA_double`
- `SHORT REAL` => `CORBA_float`
- `LONG REAL` => `ilu_longreal`

5.2.2.2 Records

Records map directly into corresponding ANSI C structures.

5.2.2.3 Unions

Because of the somewhat baroque CORBA specification, unions may take one of several forms.

Generally, ILU unions in ANSI C consist of a struct with two members: the type discriminator (a member named "_d"), and a union (a member named "_u") of the possible values. In a simple ISL union that does not name the elements, the union member names are derived from the ISL data types which compose the union. For example, if the ISL type in interface `I` is `TYPE u1 = UNION INTEGER, SHORT REAL END;` the generated ANSI C struct would be

```
typedef struct _I_u1_union I_u1;
enum I_u1_allowableTypes {
  I_u1_integer,
  I_u1_shortreal
};
struct _I_u1_union {
  enum I_u1_allowableTypes _d;
  union {
    ilu_integer integer;
    ilu_shortreal shortreal;
  } _u;
};
```

Note the discriminator `_d` may take on the values of `I_u1_integer` or `u_u1_shortreal` indicating how to interpret the data in the union. Also note how the enumerated names are formed: with the interface name and the type name prepended to the enumeration element name.

In more complex union forms, the user may specify the type of the discriminator as well as the member names and which member corresponds to which discriminator value. Consider the following ISL example:

```
INTERFACE I;
TYPE e1 = ENUMERATION red, blue, green, yellow, orange END;
TYPE u1 = e1 UNION
 a : INTEGER = red, green END,
 b : SHORT REAL = blue END,
 c : REAL
END;
```

The generated union is:

```
typedef struct _I_u1_union I_u1;
typedef enum {
  I_red = 0,
  I_blue = 1,
```

```
    I_green = 2,
    I_yellow = 3,
    I_orange = 4
  } I_e1;
  struct _I_u1_union {
    I_e1 _d;
    union {
      ilu_integer a;
      ilu_shortreal b;
      ilu_real c;
    } _u;
  };
```

This example shows that the discriminator type is to be `I_e1` and that the member names are to be `a`, `b`, and `c`. When the discriminator has the value `I_red` or `I_green` the member `a` has a valid value and the type is interpreted to be integer. When the discriminator has the value `I_green` the member `b` has a valid value and the type is interpreted to be shortreal. If the discriminator has any other value, the member `c` is expected to have a valid value and the type is interpreted to be ilu_real (double).

Discriminator types may be `INTEGER`, `ENUMERATION`, or `SHORT INTEGER`. The default for an unspecified discriminator is `SHORT INTEGER`.

5.2.2.4 Floating Point Values

The ISL `SHORT REAL` primitive type maps to the ANSI C `float` data type while `REAL` maps to `double`. The ISL `LONG REAL` primitive type currently doesn't map to anything real.

5.2.2.5 Sequences

Sequence type names, as most type definitions, are formed with the interface name and the type name. Sequence instances are represented to the ANSI C programmer as a pointer to the sequence descriptor structure. For each sequence type declared in the interface description, a pseudo-object sequence type is defined in ANSI C. These sequence types will hold any number of values of type sequence's *primary type*. For the sequence
`INTERFACE I;`
`TYPE` *T2* `= SEQUENCE OF` *T1*`;`
the following functions are defined:

I_T2 *I_T2*_**Create** (`OPTIONAL(unsigned long)` *length*, `OPTIONAL(`*T1* `*)` *initial-values*) [ANSI C]

This function creates and returns an instance of *T2*. If *length* is specified, but *initial-values* is not specified, enough space for *length* values of type *T1* is allocated in the sequence. If *initial-values* is specified, *length* is assumed to be the number of values pointed to by *initial-values*, and must be specified. Note that if type *T1* is a `character` or `short character` type, a pointer to a NIL-terminated sequence will be returned; otherwise, a normal CORBA sequence structure will be returned by value.

`CORBA_unsigned_long` *I_T2*_**Length** (*I_T2* * *s*) [ANSI C]

Returns the length of *s*.

`void` *I_T2*_**Append** (*I_T2* * *s*, *T1 value*) [ANSI C]

Appends *value* to the end of *s*. This function will reallocate space and copy, if necessary.

`void` *I_T2*_**Push** (*I_T2* * *s*, *T1 value*) [ANSI C]

Pushes *value* on to the beginning of the sequence. This function will reallocate space and copy, if necessary.

`void` *I_T2*_**Pop** (*I_T2* * *s*, *T1* * *value-ptr*) [ANSI C]

Removes the first value from the sequence *s*, and places it in the location pointed to by *value-ptr*.

`void` *I_T2*_**Every** (*I_T2* * *s*, `void (*`*func*`)(`*T1*`, void *), void *` *data*) [ANSI C]

Calls the function *func* on each element of *s* in sequence, passing *data* as the second argument to *func*.

I_T1 * *I_T2*_**Nth** (*I_T2* * *s*, `CORBA_unsigned_long` *n*) [ANSI C]

Returns the address of the *n*th element of the sequence *s*. Returns `ILU_NIL` if *n* is out of range.

`void` *I_T2*_**Init** (*I_T2* * *s*, `OPTIONAL(CORBA_unsigned_long)` *length*, `OPTIONAL(`*T1* `*)` *initial-values*) [ANSI C]

This function works like *T2*`_Create`, except that it takes a the address of an already-existing *T2* to initialize. This can be used to initialize instances of *T2* that have been stack-allocated.

`void` *I_T2*_**_Free** (*I_T2* * *s*) [ANSI C]

Frees allocated storage used internally by *s*. Does not free *s* itself.

String sequences (`SEQUENCE OF SHORT CHARACTER` or `SEQUENCE OF CHARACTER`) are just arrays of the character codes for the characters, using either Latin-1 codes (for `SEQUENCE OF SHORT CHARACTER`), or ISO 10646 Unicode codes (for `SEQUENCE OF CHARACTER`). These sequences are terminated with a character code of zero. The terminating code is not counted in the length of the sequence. All other sequence types have a record structure, mandated by CORBA:

```
typedef struct I_T2 {
  unsigned long _maximum;
  unsigned long _length;
  long *_buffer;
} I_T2;
```

The field `_maximum` contains the number of elements pointed to by `_buffer`. The field `_length` indicates the number of valid or useful elements pointed to by `_buffer`.

For example, the ISL specification

```
INTERFACE I;

TYPE iseq = SEQUENCE OF INTEGER;
```

would have in its ANSI C mapping the type

```
typedef struct I_iseq {
  unsigned long _maximum;
  unsigned long _length;
  ilu_integer *_buffer;
} I_iseq;
```

In a client program, a pointer to this type would be instantiated and initialized by calling the type specific sequence creation function generated for the sequence, e.g.

```
        ...
    I_O h;
    ILU_C_ENVIRONMENT s;
    I_iseq sq;
        ...
    sq = I_iseq_Create (0, NULL);
    I_iseq_Append (&sq, 4);
        ...
```

5.2.3 Objects and Methods

As indicated earlier, method names are generated by prepending the interface name and the class name to the method name. The first argument to a method is an object instance. The object instance is an opaque pointer value returned from a class specific constructor function. All object types are subtypes for the type defined by `ILU_C_OBJECT`, a macro which expands to the appropriate CORBA object type for the version of CORBA being used. CORBA also specifies that the type of the handle be called *interface-name_type-name*. A typedef of the CORBA-specified name to the `ILU_C_OBJECT` type is therefore generated for each object type. In the example above, the type of the object instance would be `I_O`.

Two binding procedures are specified for each object type. A *binding procedure* is a procedure that takes some name for an object instance, and returns the actual instance. Users of a module typically use a surrogate-side binding procedure, which takes the string binding handle of the object, and the most specific type ID of the object's type (if known). Suppliers of a module typically bind objects with a creation procedure, which takes an instance ID, a server on which to maintain the object, and arbitrary user data, and creates and returns the true instance of the object.

In general, for any object type *T*, the following ANSI C functions are defined:

`OPTIONAL(`*T*`)` *T*__**CreateTrue** (`OPTIONAL(RETAIN(char *))` *instance-id*, `OPTIONAL(GLOBAL(ilu_Server))` *server*, `OPTIONAL(PASS(void *))` *user-data*) [ANSI C]

Creates a true instance of type *T*, exporting it with instance-id *instance-id*, exporting it via server *server-id*, associating the value *user-data* with it. If *instance-id* is not specified, a server-relative instance-id will be assigned automatically. If *server* is not specified, the value of `ILU_C_DefaultServer` will be used, if bound. If *server* is not specified, and `ILU_C_DefaultServer` is not specified, a `NULL` pointer will be returned.

`OPTIONAL(`*T*`)` *T*__**CreateFromSBH** (`RETAIN(char *)` *sbh*, `RETAIN(CORBA_Environment *)` *Env*) [ANSI C]

Finds or creates an instance of *T*, using the given object reference.

`extern ilu_Class` *T*__**MSType** [ANSI C]

A value of type `ilu_Class` which identifies the most specific ILU type of the type *T*.

In the following example, the ILU definition is:

```
INTERFACE I;

TYPE T = OBJECT
  METHODS
    M ( r : REAL ) : INTEGER
  END;
```

This definition defines an interface `I`, an object type `T`, and a method `M`. The method `M` takes a `REAL` as an argument and returns an `INTEGER` result. The generated ANSI C header file would include the following statements:

```
typedef ILU_C_OBJECT I_T;

I_T I_T__CreateTrue (ilu_string ih, ilu_Server server, void *user_data);
I_T I_T__CreateFromSBH (char *sbh, ILU_C_ENVIRONMENT *Env);

ilu_integer I_T_M (I_T, ilu_real, ILU_C_ENVIRONMENT *);
```

The functions `I_T__CreateTrue` and `I_T__CreateFromSBH` are used to create instances of the class `I_T`. `I_T__CreateTrue` is used by servers while `I_T__CreateFromSBH` is used by clients. The pointer returned in each case is the object instance and must be passed with each method invocation.

In addition to its specified arguments, the method `I_T_M` takes an instance of the type `I_T` and a reference to a variable of type `ILU_C_ENVIRONMENT *`, which is a macro defined to be the appropriate CORBA environment type, and is used to return exception codes. The environment struct pointed to by the environment argument must be instantiated in a client; its address is passed as the last argument to each method. True procedures must expect a pointer to this structure as the last argument.

Finally, the ANSI C client calling the method for `M` might be as follows:

```
#include "I.h"

int main (int ac, char **av)
{
  double atof( );
  I_T inst;
  int xx;
  double f;
  ILU_C_ENVIRONMENT ev;

  I__Initialize( );
  f = atof (av[1]);
  inst = I_T__CreateFromSBH (av[2], &ev);
  if (!ILU_C_SUCCESSFUL(&ev)) {
    printf( "CreateFromSBH raised exception <%s>\n",
      ILU_C_EXCEPTION_ID(&ev));
    return(1);
  }
  xx = I_T_M (inst, f, &ev);
  if (!ILU_C_SUCCESSFUL(&ev)) {
    printf( "exception <%s> signalled on call to I_T_M\n",
      ILU_C_EXCEPTION_ID(&ev));
    return(2);
  }
  printf( "result is %d\n", xx );
  return(0);
}
```

Note the call on the interface-specific client initialization procdedure `I__Initialize`; these are described in a later section.

In this example, the string binding handle is obtained from standard input along with some floating-point value.

The class specific function `I_T__CreateFromSBH` is called to obtain the object instance. This function was passed the string binding handle, and a CORBA environment in which to report exceptions. The returned object instance is then passed as the first argument to the method `I_T_M`, along with the environment `ev`, and the single actual `ilu_real` argument `f`. `I_T_M` returns an `ilu_integer` value which is placed in `xx`.

The true implementation of the method `M` might be as follows:

```
ilu_integer server_I_T_M ( I_T h, ilu_real u, ILU_C_ENVIRONMENT *s )
{
  return( (ilu_integer) (u + 1) );
```

```
}
```

In this simple example, the corresponding server, or true, method computes some value to be returned. In this case it adds one to its `ilu_real` argument `u`, converts the value to an integer, and returns that value. Note that the server method, if not signalling any exceptions, may ignore the environment parameter.

5.2.3.1 Interface Inheritance

Through interface inheritance, an object type may participate in the behaviors of several different types that it inherits from. These types are called

ancestors of the object type. In ANSI C, an object type supplies all methods either defined directly on that type, or on any of its ancestor types.

Consider the following example:

```
INTERFACE I2;

EXCEPTION E1;

TYPE T1 = OBJECT
  METHODS
    M1 (a : ilu.CString) : REAL RAISES E1 END
  END;

TYPE T2 = OBJECT
  METHODS
    M2 ( a : INTEGER, Out b : INTEGER )
  END;

TYPE T3 = OBJECT SUPERTYPES T1, T2 END
  METHODS
    M3 ( a : INTEGER )
  END;
```

The object type `T3` inherits from the object type `T2`. Thus, five ANSI C methods are relevant to the interface `I2`: `server_I2_T1_M1`, `server_I2_T2_M2`, `server_I2_T3_M1`, `server_I2_T3_M2`, and `server_I2_T3_M3`. A module that implements true instances of `T3` would have to define the last three true methods. A client uses only three generic functions: `I2_T1_M1`, `I2_T2_M2`, and `I2_T3_M3`.

5.2.3.2 Object Implementation

This information is provided for those interested in the implementation of the ANSI C object system. It is **not** *guaranteed to remain the same from release to release.*

Each object type is represented by a *TypeVector*, which is a vector of pointers to *MethodBlock* structs, one for each component type of the object type, ordered in the proper class precedence for that object type. Each MethodBlock struct contains a `ilu_Class` value, followed by a vector of pointers to the methods directly defined by that `ilu_Class`. There are two different TypeVectors for each object type, one for the surrogate class of the type, and the other for the true class of the type. The TypeVector for the surrogate class uses the MethodBlocks of its supertypes; the TypeVector for the true class uses its own MethodBlocks for both direct and inherited methods, as true classes in the ANSI C implementation override all of their methods. The TypeVectors, and MethodBlocks for true classes, are not exported; the MethodBlocks for surrogate classes are, as they are used by their subclasses.

For each method directly defined in the type, a generic function is defined in the common code for its interface, which dispatches to the appropriate method. It does this by walking down the TypeVector for the object, till it finds a MethodBlock which contains the appropriate ilu_Class on which this method is directly defined), then calling the method pointer which is indexed in the MethodBlock's vector of method pointers by the index of the method. The generic functions have the correct type signature for the method. They can be referenced with the `&` operator.

5.2.4 Exceptions

ANSI C has no defined exception mechanism. As already indicated, exceptions are passed in ILU ANSI C by adding to the end of each method an additional status argument that can convey an exception code and a value of a type associated with that exception. To signal an exception, a method implementation sets the exception code and supplies the parameter value (if any).

In the following example, the `div` method can raise the exception `DivideByZero`:

```
INTERFACE calc;

TYPE numerator = INTEGER;

EXCEPTION DivideByZero : numerator;

TYPE self = OBJECT
  METHODS
    Div( v1 : INTEGER, v2 : INTEGER ) : INTEGER RAISES DivideByZero END
  END;
```

The generated include file, calc.h contains the exception definitions:

```
#ifndef __calc_h_
#define __calc_h_
/*
** this file was automatically generated for C
** from the interface spec calc.isl.
*/
```

```
#ifndef __ilu_c_h_
#include "ilu-c.h"
#endif

extern ILU_C_ExceptionCode    _calc__Exception_DivideByZero;
#define ex_calc_DivideByZero _calc__Exception_DivideByZero

typedef ilu_integer calc_numerator;
typedef calc_numerator calc_DivideByZero;

typedef ILU_C_OBJECT calc_self;

calc_self calc_self__CreateTrue ( char *id, ilu_Server server,
   void * user_data);
calc_self calc_self__CreateFromSBH ( char * sbh, ILU_C_ENVIRONMENT *Env );

ilu_integer calc_self_Div( calc_self, ilu_integer, ilu_integer,
   ILU_C_ENVIRONMENT *Env );

extern void calc__BindExceptionValue (ILU_C_ENVIRONMENT *, ilu_Exception, ...);

#endif
```

The method implementation for `Div` in the true module must detect the divide-by-zero condition and raise the exception:

```
long server_calc_self_Div (calc_self h, ilu_integer u, ilu_integer v, ILU_C_ENVIRONMENT
*s)
{
  calc_numerator n = 9;

  if ( v == 0 )
    {
      s->_major = ILU_C_USER_EXCEPTION;
      s->returnCode = ex_calc_DivideByZero;
      s->ptr = (void *) malloc(sizeof(calc_numerator));
      *((calc_numerator *) (s->ptr)) = n;
      s->freeRoutine = (void (*) (void *)) 0;
      return( u );
    }
  else
    return( u / v );
}
```

When freeing the parameter requires more than just freeing `s->ptr`, a non-NULL `s->freeRoutine` is provided that does the additional freeing; `s->freeRoutine` is given one argument, `s->ptr`, and returns `void`.

The generated stubs offer as a convenience a variadic procedure that can be used to raise any exception declared in the interface. Using this procedure, the above code would be:

```
long server_calc_self_Div (calc_self h, ilu_integer u, ilu_integer v, ILU_C_ENVIRONMENT
*s)
{
  calc_numerator n = 9;

  if ( v == 0 )
    {
      calc__BindExceptionValue(s, ex_calc_DivideByZero, n);
      return( u );
    }
  else
    return( u / v );
}
```

The exception is sent back to the client, which can detect it thusly:

```
  ...
  calc_self instance;
  ILU_C_ENVIRONMENT s;
  ilu_integer i, j;
  ilu_integer val;
  ...
  instance = calc_self__CreateFromSBH (sbh, &s);

  if (! ILU_C_SUCCESSFUL(&s)) {
    fprintf (stderr, "CreateFromSBH(%s) raised %s\n",
      sbh, ILU_C_EXCEPTION_ID (&s) );
    exit(1);
  }

  val = calc_self_Div (instance, i, j, &s);

  /* check to see if an exception occured */

  if (! ILU_C_SUCCESSFUL(&s)) {
    /* report exception to user */
    char *p;

    p = ILU_C_EXCEPTION_ID (&s);

    if (p == ex_calc_DivideByZero) {
      calc_numerator *ip;
      ip = (calc_numerator *) ILU_C_EXCEPTION_VALUE (&s);
      fprintf (stderr, "%s signaled:  numerator = %d\n", p, *ip);
      }
    else {
      /* odd exception at this point */
```

```
        fprintf (stderr, "Unexpected <%s> on call to Div.\n", p);
        }
      /* free up any transient exception data */
      ILU_C_EXCEPTION_FREE (&s);
      }
    else {
      /* no exception - print the result */
      printf( "result is %d \n", val );
      }
    ...
```

5.2.5 True Module (Server Module) Construction

This section will outline the construction of a true module exported by an address space. For the example, we will demonstrate the calculator interface described above. We will also use the CORBA 2.0 names for standard types and exceptions, to show that it can be done.

First, some runtime initialization of the server stubs must be done. Call *Foo*`__InitializeServer` for every ISL interface *Foo* containing an object type implemented by the address space. Also call any client initialization procedures needed (see next section). These server and client initialization calls can be made in any order, and each initialization procedure can be called more than once. However, no two calls may be done concurrently (this is an issue only for those using some sort of multi-threading package).

Then we create an instance of `calc_self`. We then make the string binding handle of the object available by printing it to stdout. Finally the `ILU_C_Run` procedure is called. This procedure listens for connections and dispatches server methods.

The main program for the server is as follows:

```
#include "I2.h"

CORBA_long
  server_calc_self_Div (calc_self h,
                        CORBA_long u,
                        CORBA_long v,
                        CORBA_Environment *s)
{
  calc_numerator n = 9;

  if ( v == 0 )
    {
      calc__BindExceptionValue(s, ex_calc_DivideByZero, n);
      return( u );
    }
  else
    return( u / v );
}
```

```
main ()
{
  calc_self s;
  char * sbh;
  CORBA_Environment ev;

  calc__InitializeServer( );

  s = calc_self__CreateTrue (NULL, NULL, NULL);
  if (s == NULL)
    {
      fprintf (stderr, "Unable to create instance of calc_self.\n");
      exit(1);
    }
  else
    {
      sbh = CORBA_ORB_object_to_string (ILU_C_ORB, &ev, s);
      if (ev._major == CORBA_NO_EXCEPTION)
        {
          printf ("%s\n", sbh);
          ILU_C_Run (); /* enter main loop; hang processing requests */
        }
      else
        {
          fprintf (stderr,
                   "Attempt to obtain sbh of object %p signalled <%s>.\n",
                   s, CORBA_exception_id(&ev));
          exit(1);
        }
    }
}
```

5.2.6 Using ILU Modules

Before manipulating surrogate objects, a client module must first call a runtime initialization procedure *Foo*`__Initialize` for each ISL interface *Foo* that declares object types whose surrogates are to be manipulated. Additionally, server modules must also call server initialization procedures (see previous section). These initialization calls may be made in any order, and each procedure may be called more than once. However, no two calls may be done concurrently (this is an issue only for those using some sort of multi-threading package).

A client of an exported module may obtain an object instance either by calling a method which returns the instance, or by calling *TYPE*`__CreateFromSBH()` on the string binding handle of an instance. Once the object instance, which is typically a surrogate instance, but may in fact be a true instance, is held by the client, it can be used simply by making method calls on it, as shown above.

5.2.7 Stub Generation

To generate ANSI C stubs from an ISL file, use the program c-stubber. Four files are generated from the '`.isl`' file:

- '*interface-name*`.h`' contains the definitions for the types and procedures defined by the interface and used by the generated stubs.
- '*interface-name*`-common.c`' contains the general code used by both client and server; and
- '*interface-name*`-surrogate.c`' contains the client-side and general code for the interface; and
- '*interface-name*`-true.c`' contains the server-side stubs and code for the interface.

Typically, clients of a module never have a need for the '*interface-name*`-true.c`' file.

```
% c-stubber foo.isl
header file interface foo to ./foo.h...
code for interface foo to ./foo-common.c...
code for interface foo to ./foo-surrogate.c...
code for server stubs of interface foo to ./foo-true.c...
%
```

5.2.8 Tailoring Identifier Names

The option `-renames` *renames-filename* may be used with `c-stubber` to specify particular ANSI C names for ISL types.

It is sometimes necessary to have the ANSI C names of an ILU interface match some other naming scheme. A mechanism is provided to allow the programmer to specify the names of ANSI C language artifacts directly, and thus override the automatic ISL to ANSI C name mappings.

To do this, you place a set of synonyms for ISL names in a

renames-file, and invoke the `c-stubber` program with the switch `-renames`, specifying the name of the renames-file. The lines in the file are of the form

construct ISL-name ANSI C-name

where *construct* is one of `method`, `exception`, `type`, `interface`, or `constant`; *ISL-name* is the name of the construct, expressed either as the simple name, for interface names, the concatenation *interface-name*`.`*construct-name* for exceptions, types, and constants, or *interface-name*`.`*type-name*`.`*method-name* for methods; and *ANSI C-name* is the name the construct should have in the generated ANSI C code. For example:

```
# change "foo_r1" to plain "R1"
type foo_r1 r1
# change name of method "m1" to "method1"
method foo_o1_m1 method1
```

Lines beginning with the 'sharp' character '`#`' are treated as comment lines, and ignored, in the renames-file.

This feature of the `c-stubber` should be used as little and as carefully as possible, as it can cause confusion for readers of the ISL interface, in trying to follow the ANSI C code. It can also create name conflicts between different modules, unless names are carefully chosen.

5.3 Libraries and Linking

For clients of an ILU module, it is only necessary to link with the '*interface-name*`-surrogate.o`' and '*interface-name*`-common.o`' files generated from the ANSI C files generated for the interface or interfaces being used, and with the two libraries '*ILUHOME*`/lib/libilu-c.a`' and '*ILUHOME*`/lib/libilu.a`' (in this order, as '`libilu-c.a`' uses functions in '`libilu.a`').

For implementors of servers, the code for the server-side stubs, in the file '*interface-name*`-true.o`' compiled from '*interface-name*`-true.c`', and in the file '*interface-name*`-common.o`' compiled from '*interface-name*`-common.c`', should be included along with the other files and libraries.

5.4 ILU C API

In addition to the functions defined by the CORBA mapping, the ILU ANSI C mapping provides some other functions, chiefly for type manipulation, object manipulation, and server manipulation. There are also a number of macros provided for compatibility with both versions of CORBA (revision 2.0).

5.4.1 Type Manipulation

`OPTIONAL(ilu_Class)` **ILU_C_FindILUClassByTypeName** (`RETAIN(ilu_string)` *type-name*) [ILU C API]

Locking: L1_sup < otmu, L2, Main unconstrained.

Given the *type-name* of an ILU object type, of the form `"Interface.Typename"`, returns the `ilu_Class` value for it. This value can be used to compare types for equality.

`OPTIONAL(ilu_Class)` **ILU_C_FindILUClassByTypeID** (`RETAIN(ilu_string)` *type-id*) [ILU C API]

Locking: L1_sup < otmu; L2, Main unconstrained.

Given the *type-id* of an ILU object type, of the form `"ilu:gfbSCM7tsK9vVYjKfLole1HOBDc"`, returns the `ilu_Class` value for it. This value can be used to compare types for equality.

`GLOBAL(OPTIONAL(ilu_string))` **ILU_C_ClassName** (`RETAIN(CORBA_Object)`) [ILU C API]

Locking: unconstrained.

Returns the ILU name for the most specific type of an object instance.

`GLOBAL(OPTIONAL(ilu_string))` **ILU_C_ClassID** (`RETAIN(CORBA_Object)`) [ILU C API]

Locking: unconstrained.

Returns the ILU type ID for the most specific type of an object instance.

`ilu_Class` **ILU_C_ClassRecordOfInstance** (`CORBA_Object`) [ILU C API]

Locking: unconstrained.

Returns the `ilu_Class` value for the most specific type of an object instance.

5.4.2 Object Manipulation

`ilu_string` **ILU_C_SBHOfObject** (`CORBA_Object` *instance*) [ILU C API]

Locking: Main invariant holds.

Given an *instance*, returns a reference to that instance. The CORBA-specified routine `CORBA_ORB_object_to_string()` should typically be used instead.

`OPTIONAL(CORBA_Object)` **ILU_C_SBHToObject** (`char *` *sbh*, `ilu_Class` *static_type*, `RETAIN(CORBA_Environment *) Env`) [ILU C API]

Locking: Main invariant holds.

Takes an object reference and returns the object. *static_type* is a type the caller knows the object to have.

`OPTIONAL(PASS(char*))` **ILU_C_PublishObject** (`CORBA_Object` *instance*) [ILU C API]

Locking: Main invariant holds.

Publishes the OID of the *instance* in a domain-wide registry. This is an experimental interface, and may change in the future.

`ilu_boolean` **ILU_C_WithdrawObject** (`CORBA_Object` *instance*, `PASS(char *)` *proof*) [ILU C API]

Locking: Main invariant holds.

Removes the OID of the *instance* from the domain-wide registry. *proof* is the string returned from the call to `ILU_C_PublishObject()`.

`OPTIONAL(GLOBAL(CORBA_Object))` **ILU_C_LookupObject** ([ILU C API]
`RETAIN(char *)` *sid*, `RETAIN(char *)` *ih*, `ilu_Class` *static-class*)

Locking: Main invariant holds.

Using the local registry, find and return the object specified by the given Server ID and server-relative Instance Handle. *static_type* is one you know the actual object must have; it may also have more refined types. For an already-reified surrogate this procedure will reconsider what contact info to use for reaching the server.

5.4.3 Server Manipulation

`void` **ILU_C_Run** (`void`) [ILU C API]

Locking: Main invariant holds.

Called to animate a server and/or other parts of the program. Used only in single-threaded mode. Invokes the event handling loop. Never returns.

`ilu_Server` **ILU_C_InitializeServer** (`OPTIONAL(RETAIN(char *))` [ILU C API]
serverID, `OPTIONAL(GLOBAL(ilu_ObjectTable))` *obj_tab*,
`OPTIONAL(RETAIN(char *))` *protocol*, `OPTIONAL(RETAIN(ilu_TransportInfo))`
transport, `ilu_boolean` *setdefaultport*)

Locking: Main invariant holds.

Creates and returns an `ilu_Server` with ID *serverID*, object mapping table *obj_tab*, using protocol *protocol* over a transport stack specified by *transport*. If *serverID* is specified as `NULL`, a unique string is generated automatically for the server ID. If *obj_tab* is specified as `NULL`, the default hash table object table is used.

If either *protocol* or *transport* is specified, or if *setdefaultport*, an `ilu_Port` will automatically be created and added to the `ilu_Server`. *protocol*, if not `NULL`, is a string that specifies which RPC protocol to use on the port; `NULL` causes use of `Sun RPC`. *transport*, if not `NULL`, is a sequence of strings that specifies the transport stack to use below the RPC protocol; `NULL` signifies use of SunRPC Record Marking over TCP to/from one of the IP addresses of this host. Chapter 8 gives details on protocol and transport strings. If *setdefaultport* is true, the newly created `ilu_Port` will become the default port of the `ilu_Server`.

5.4.4 CORBA Compatibility Macros

ILU supports CORBA 2.0, and formerly supported either 1.1 or 1.2, depending on how it was installed at your site. A number of macros are defined to make programs less dependent on which version they use.

ILU_C_OBJECT Macro

Expands to `CORBA_Object`.

ILU_C_ENVIRONMENT Macro

Expands to `CORBA_Environment`.

ILU_C_NO_EXCEPTION Macro

Expands to `CORBA_NO_EXCEPTION`.

ILU_C_USER_EXCEPTION Macro

Expands to `CORBA_USER_EXCEPTION`.

ILU_C_SYSTEM_EXCEPTION Macro

Expands to `CORBA_SYSTEM_EXCEPTION`.

ILU_C_SUCCESSFUL (*ILU_C_ENVIRONMENT * ev*) Macro

Evaluates to true if no exception has been raised.

ILU_C_SET_SUCCESSFUL (*ILU_C_ENVIRONMENT * ev*) Macro

Sets *ev* to a successful result.

ILU_C_EXCEPTION_ID (*ILU_C_ENVIRONMENT * ev*) Macro

Returns the `char *` value that is the exception's ID.

ILU_C_EXCEPTION_VALUE (*ILU_C_ENVIRONMENT * ev*) Macro

Expands to `CORBA_exception_value(`*ev*`)`.

ILU_C_EXCEPTION_FREE (*ILU_C_ENVIRONMENT * ev*) Macro

Expands to `CORBA_exception_free(`*ev*`)`.

6 Using ILU with Modula-3

This document is for the Modula-3 programmer who wishes to use ILU. ILU currently supports only DEC SRC Modula-3 version 2.08.

6.1 Mapping ILU ISL to Modula-3

6.1.1 Names

An item named `Bar` in ISL interface `Foo` becomes an item named `Bar` in the Modula-3 interface `Foo`. A hyphen in an ISL name becomes an underscore in the corresponding Modula-3 name.

6.1.2 Types

ISL types appear in Modula-3 as follows:

1. `SHORT INTEGER` becomes `[-32768 .. 32767]`.
2. `INTEGER` becomes `INTEGER`.
3. `LONG INTEGER` becomes

   ```
   TYPE LongInt = RECORD
                    high: [-16_80000000 .. 2147483647];
                    low : Word.T (*[0 ..  4294967295]*)
                  END;
   ```

   This represents the number `high*2^32 + low`. We always have the invariants `-2^31 <= high < 2^31` and `0 <= low < 2^32`, even on systems whose natural word size is greater than 32 bits.
4. `BYTE` becomes `[0 .. 255]`.
5. `SHORT CARDINAL` becomes `[0 .. 65535]`.
6. `CARDINAL` becomes `Word.T`.
7. `LONG CARDINAL` becomes `RECORD high, low: Word.T END`. This representation works analogously to that for `LONG CARDINAL`.
8. `SHORT REAL` becomes `REAL`.
9. `REAL` becomes `LONGREAL`.
10. `LONG REAL` becomes an opaque type. Values of this type can only be handed around; no other operations are provided, not even equality testing. `LONG REAL` is not really supported yet.
11. `SHORT CHARACTER` becomes `['\000' .. '\377']`.
12. `CHARACTER` becomes `[0 .. 65535]`.
13. Variable-length `ARRAY`s of `SHORT CHARACTER` become `TEXT`.
14. Other variable-length arrays become `REF ARRAY OF`.
15. Fixed-length arrays of `SHORT CHARACTER` become arrays of `BITS 8 FOR ['\000' .. '\377']`.

16. Fixed or variable-length ARRAYs of BYTE become arrays of BITS 8 FOR [0 .. 255].
17. No other arrays specify packing in the Modula-3.
18. A fixed length array, ARRAY OF *L1*, ... *Ln*, becomes ARRAY [0 .. *L1*-1] OF ... ARRAY [0 .. *Ln*-1] OF.
19. An ISL record becomes a M3 record.
20. An ISL union becomes a M3 object type and some subtypes. The ISL

```
TYPE Foo = DiscT UNION
    case1: T1 = val1-1, ... val1-j END,
    ...
    casen: Tn = valn-1, ... valn-k END
    END OTHERS;
```

maps to the Modula-3

```
TYPE Foo = BRANDED OBJECT d: DiscT END;
TYPE  Foo_case1 = Foo BRANDED OBJECT v: T1 END;
CONST Foo_case1__Code : DiscT = val1-1;
...
TYPE  Foo_casen = Foo BRANDED OBJECT v: Tn END;
CONST Foo_casen__Code : DiscT = valn-1;
TYPE  Foo_OTHERS = Foo BRANDED OBJECT END;
(* Where every Foo is of one of the subtypes enumerated here,
   and the tag field (d) is consistent with the subtype. *)
```

The *Foo*_OTHERS subtype appears only for union constructions including the OTHERS keyword. If the ISL union has a DEFAULT arm

```
    cased: Td = DEFAULT
```

it maps to another subtype in Modula-3:

```
TYPE  Foo_cased = Foo BRANDED OBJECT v: Td END;
```

The *Foo*_*casen*__Code constants are conveniences for filling in and decoding the d field. Note that code that creates a *Foo* is responsible for filling in the d field.

21. An ISL enumeration becomes a M3 enumeration. Due to the fact that Modula-3 offers no way to specify the codes used to represent enumerated values, the codes specified in ISL, if any, have no effect on the translation.
22. When a *Foo* becomes a *Bar*, an OPTIONAL *Foo* becomes a REF *Bar*, unless *Bar* is a subtype of REFANY, in which case OPTIONAL *Foo* becomes *Bar*; NIL encodes the NULL case.
23. An ISL object type becomes a Modula-3 object type. The ISL adjectives SINGLETON, DOCUMENTATION, COLLECTIBLE, OPTIONAL, AUTHENTICATION, and BRAND have no effect on the mapping into the Modula-3 type system.

    OUT and INOUT method parameters in ISL become VAR parameters in Modula-3; IN parameters become VALUE (by default) parameters. The SIBLING constraint in ISL has no manifestation in the Modula-3 type system.

The methods are declared to raise the exceptions `IluBasics.Failed` and `Thread.Alerted` in addition to the exceptions declared in the ISL. Exception `IluBasics.Failed` is used to convey all the errors that can arise from the RPC mechanism, except `Thread.Alerted`. Is the surrogate (and the other surrogates from the same server?) broken after either of these exceptions is raised?

Because ILU has multiple inheritance (i.e., an object type can have more than one direct supertype), the Modula-3 subtype relation is a sub-relation of the ILU subtype relation. In general, an ILU object type is mapped to a suite of Modula-3 object types, and a cohort of Modula-3 objects (one of each of the suite of Modula-3 types) correspond to one ILU object. There will be only one Modula-3 object (type) when only single-inheritance is used in constructing the ILU object type: when every ancestor type has at most one direct ancestor. Except where the programmer knows this is the case, and plans for it to remain so, she must abandon the native Modula-3 `TYPECASE`/`NARROW`/automatic-widen facilities for explicit calls that invoke the ILU subtype relation.

To generalize the Modula-3 `TYPECASE`/`NARROW`/automatic-widen facilities, the Modula-3 object type `Ilu.Object` includes the following method:

```
PROCEDURE ILU_Qua_Type(ot: ObjectType): Object;
```

If the object has, in ILU, the given object type, the Modula-3 object of the appropriate Modula-3 type is returned; otherwise, NIL is returned. As an added convenience, the Modula-3 mapping of interface *Foo* will contain, for each of its object types *Bar*:

`PROCEDURE ILU_Qua_`*Bar*`(x: Ilu.Object): `*Bar*`;`

This procedure takes a non-`NIL` argument. If the argument is, in ILU, an instance of *Bar* or one of its subtypes, the corresponding language-specific object is returned; otherwise, `NIL` is returned.

6.1.3 Exceptions

ISL exceptions are exactly like Modula-3 exceptions, and are mapped directly.

6.1.4 Example

Here's a sample ISL spec, and the resulting Modula-3 mappings:

```
INTERFACE Foo;

TYPE String = ilu.CString;
TYPE UInt = CARDINAL;

TYPE E1 = ENUMERATION val1, val2, val3 = 40 END;
TYPE R1 = RECORD field1 : CARDINAL, field2 : E1 END;
TYPE FAB = ARRAY OF 200 BYTE;
TYPE VAB = SEQUENCE OF BYTE;
```

```
TYPE FASC = ARRAY OF 10 SHORT CHARACTER;
TYPE VASC = SEQUENCE OF SHORT CHARACTER;
TYPE FAC = ARRAY OF 5 CHARACTER;
TYPE VAC = SEQUENCE OF CHARACTER;
TYPE A2 = ARRAY OF 41, 3 R1;
TYPE S1 = SEQUENCE OF E1;
TYPE U1 = UNION R1, A2 END;

EXCEPTION Except1 : String;

CONSTANT Zero : CARDINAL = 0;

TYPE O1 = OBJECT
    METHODS
        M1(r1: R1, INOUT v: VASC, OUT s1: S1): UInt RAISES Except1 END,
        FUNCTIONAL Hash(v: VASC): FASC,
        ASYNCHRONOUS Note(x: LONG REAL)
    END;
```

The Modula-3 mapping:

```
INTERFACE Foo;

IMPORT Ilu, IluBasics, Thread;
IMPORT ilu; <*NOWARN*>

TYPE UInt = CARDINAL;
TYPE E1 = {
  val1,
  val2,
  val3};
TYPE R1 = RECORD
   field1 : CARDINAL;
   field2 : E1;
END;
TYPE VASC = TEXT;  (* NIL not allowed *)
TYPE S1 = REF ARRAY OF E1;  (* NIL not allowed *)
TYPE FASC = ARRAY [0..9] OF Ilu.PackedShortChar;


(* declaration of M3 type "Foo.O1" from ILU class "Foo:O1"  *)

TYPE O1 = Ilu.Object OBJECT
  METHODS
    M1 (r1: R1; VAR v: VASC; VAR s1: S1): UInt
       RAISES {IluBasics.Failed, Thread.Alerted, Except1};
    Hash (v: VASC): FASC RAISES {IluBasics.Failed, Thread.Alerted};
    Note (x: Ilu.LongReal) RAISES {IluBasics.Failed, Thread.Alerted};
  OVERRIDES
    ILU_Get_Type := ILU_Get_Type_O1
```

```
  END;

PROCEDURE ILU_SBH_To_O1 (sbh: TEXT; mostSpecificTypeID: TEXT := NIL): O1
  RAISES {IluBasics.Failed, Thread.Alerted};

PROCEDURE ILU_Get_Type_O1 (self : Ilu.Object): Ilu.ObjectType;

PROCEDURE ILU_Qua_O1 (x: Ilu.Object): O1;

TYPE A2 = ARRAY [0..40] OF ARRAY [0..2] OF R1;
TYPE  U1 = BRANDED OBJECT d: Ilu.ShortInt END;  (* NIL not allowed *)
TYPE  U1_R1       = U1 BRANDED OBJECT v: R1 END;
CONST U1_R1__Code : [-32768..32767] = 0;
TYPE  U1_A2       = U1 BRANDED OBJECT v: A2 END;
CONST U1_A2__Code : [-32768..32767] = 1;
TYPE VAC = REF ARRAY OF Ilu.Character;  (* NIL not allowed *)
TYPE FAC = ARRAY [0..4] OF Ilu.Character;
TYPE VAB = REF ARRAY OF BITS 8 FOR Ilu.Byte;  (* NIL not allowed *)
TYPE FAB = ARRAY [0..199] OF Ilu.PackedByte;
TYPE String = TEXT;  (* NIL not allowed *)

CONST Zero : CARDINAL = 0;

(* Exceptions *)

EXCEPTION Except1 (String);

END Foo.
```

6.2 Importing an ILU interface in Modula-3

A client can acquire a Modula-3 language-specific object by calling the `ILU_SBH_To_...` stub procedure, passing the string binding handle and most specific type ID; these are typically obtained through some name service. The Simple Binding facility is available in an integrated way, as exhibited later.

The client can then proceed to make calls on the object.

6.3 Exporting an ILU interface in Modula-3

A server uses the following interface to expose itself to the ILU/M3 runtime.

```
INTERFACE Ilu;
IMPORT IluKernel, Word;
FROM IluBasics IMPORT Failed, Failure;

<*PRAGMA lL, Ll, Main*>
```

```
(* Concurrency and locking:

   As in iluExports.h.  The ILU/Modula-3 runtime adds the folloing
   mutexes:
| ssMu global mutex for server registry;
| srmu global mutex for StrongRef implementation;
| ocMu global mutex for ObjectCreator registry;
| Ilu.Server each one is a mutex;

   and the following ordering constraints:
|  IluKernel.Server < ssMu < Ilu.Server
|  IluKernel.Server < srmu
|  IluKernel.Server < ocMu

   *)

(* RPC protocol failures *)

TYPE
  ProtocolFailure =
    Failure BRANDED OBJECT case: ProtocolFailureCase; END;


  ProtocolResultCode =
    {Success, NoSuchTypeAtServer, TypeVersionMismatch,
     NoSuchMethodOnType, GarbageArguments, Unknown, LostConnection,
     RequestRejected, RequestTimeout};

  ProtocolFailureCase = [ProtocolResultCode.NoSuchTypeAtServer ..
                         ProtocolResultCode.RequestTimeout];


(* Datatypes defined in ISL. *)

TYPE Byte = [0 .. 255];
TYPE PackedByte = BITS 8 FOR Byte;
TYPE ShortInt = [-32768 .. 32767];
TYPE Integer = INTEGER;
TYPE
  LongInt = RECORD
              high: [-16_80000000 .. 2147483647];
              low : Word.T (*[0 ..  4294967295]*)
            END;
TYPE ShortCard = [0 .. 65535];
TYPE Cardinal = Word.T;
TYPE LongCard = RECORD high, low: Word.T (*[0 ..  4294967295]*) END;
TYPE ShortReal = REAL;
TYPE Real = LONGREAL;
TYPE LongReal <: REFANY;
TYPE ShortChar = ['\000' .. '\377'];
```

```
TYPE PackedShortChar = BITS 8 FOR ShortChar;
TYPE Character = ShortCard; (* In Unicode. *)
TYPE String = TEXT; (* With no embedded '\000'. *)
TYPE WString = REF ARRAY OF Character; (* With no embedded 0. *)
TYPE Bytes = REF ARRAY OF PackedByte;


(* The String Binding Handle. *)

TYPE
  SBH = TEXT;
  (* A string that includes an instance ID and a contact-info *)

TYPE
  InstanceId = TEXT;
  (* A unique identifier for an object; it is factored into a ServerId
     and an ObjectHandle. *)

TYPE
  ServerId = TEXT;
  (* A unique identifier for a server *)

TYPE
  ObjectHandle = TEXT;
  (* A server-relative identifier for an object *)

TYPE
  ContactInfo = TEXT;
  (* An encoding of how to reach a server *)


(* ================ Server stuff ================ *)

TYPE
  ServerPrivate <: ROOT;
  Server = ServerPrivate OBJECT
             <*lL, Ll, Main unconstrained*>
             id: ServerId; (*READONLY*)
           METHODS
           END;
  (* A data structure that represents a server, either local to this
     program or remote.  Each server is actually one of the following
     two types. *)

TYPE SurrogateServer <: Server;

TYPE
  TrueServer <:
    Server OBJECT
    METHODS
```

```
      <*Main Invariant holds; Ll otherwise unconstrained*>

      HandleListenerFailure (f: Failure): FailureAction;
      (* When there's a failure in a listener for this server, this
         procedure is notified, and the result indicates whether the
         listener is abandoned or continues listening. *)
      HandleWorkerFailure (f: Failure): FailureAction;
      (* When there's a failure in a worker for this server, this
         procedure is notified, and the result indicates whether the
         connection is abandoned or continues listening. *)
    END;
  (* A server local to this program. *)

TYPE FailureAction = {Quit, Continue};

<*lL, Ll = {}*>
PROCEDURE DefaultHandleListenerFailure (self: TrueServer; f: Failure):
  FailureAction;

<*lL, Ll = {}*>
PROCEDURE DefaultHandleWorkerFailure (self: TrueServer; f: Failure):
  FailureAction;

<*Main Invariant holds; Ll otherwise unconstrained*>

PROCEDURE InitTrueServer (self  : TrueServer;
                          id    : ServerId := NIL;
                          objtab: ObjectTable := NIL ): TrueServer
  RAISES {Failed};

TYPE
  ObjectTable =
    OBJECT
    METHODS
      <*lL >= {the kernel server}*>
      <*lL >= {gcmu} if the object is collectible*>
      <*Ll, Main unconstrained*>

      ObjectToHandle (o: Object): ObjectHandle;
      (* Returns the handle associated with the given object, inventing
         and recording a new handle if necessary. *)
      HandleToObject (h: ObjectHandle): Object;
      (* Returns the Object associated with the given handle, or NIL if
         no such Object. *)
    END;
  (* An one-to-one association between Objects and ObjectHandles, such
     as a server might maintain. *)

PROCEDURE Export_Server (server: TrueServer;
                         p     : ProtocolInfo;
```

```
                        t     : TransportInfo ) RAISES {Failed};

TYPE ProtocolInfo = BRANDED OBJECT END;
TYPE SunRpc2 = ProtocolInfo BRANDED OBJECT prognum, version := 0 END;
TYPE Courier = ProtocolInfo BRANDED OBJECT prognum, version := 0 END;

TYPE TransportInfo = BRANDED OBJECT END;
TYPE
  TCP = TransportInfo BRANDED OBJECT host, port := 0 END;
  UDP = TransportInfo BRANDED OBJECT host, port := 0 END;
  (* host and port are in host, not network, byte order. *)
TYPE SPP = TransportInfo BRANDED OBJECT addr := AnyXnsAddr END;

TYPE
  XnsAddr = RECORD
              net   : XnsNet;
              host  : XnsHost;
              socket: XnsSocket
            END;
  XnsNet = Cardinal;
  XnsHost = ARRAY [0 .. 5] OF PackedByte;
  XnsSocket = ShortCard;
CONST AnyXnsAddr = XnsAddr{0, XnsHost{0, ..}, 0};

TYPE Root <: ROOT;

TYPE
  Object <: ObjectPublic;
  ObjectPublic =
    Root OBJECT
      <*lL, Ll, Main unconstrained*>
      ilu_is_surrogate: BOOLEAN := FALSE;
    METHODS
      <*lL, Ll, Main unconstrained*>

      ILU_Get_Server (): Server;
      ILU_Get_Type (): ObjectType;
      (* Returns the most specific ILU type known to this program for
         the ILU object represented by this Modula-3 object. *)
      ILU_Qua_Type (ot: ObjectType): Object;

      <*Main Invariant holds; Ll otherwise unconstrained*>

      ILU_Close           () RAISES {};
      ILU_Close_Surrogate () RAISES {};
    END;

TYPE ObjectType = IluKernel.ObjectType;

PROCEDURE SbhFromObject (o: Object): SBH RAISES {Failed};
```

```
  (* May be applied to any Object; returns a reference that can be
     passed to other programs.  Export_Server must have been called on
     the object's server. *)

<*lL, Ll, Main unconstrained*>
PROCEDURE IdOfObjectType (ot: ObjectType): TEXT;
  (* Returns a shortish string that identifies this object type. *)


END Ilu.
```

A server module begins by creating an `Ilu.TrueServer` and calling `Ilu.InitTrueServer` on it. The server module may either specify the server's ID in this call, or let the ILU runtime choose one. The server module may specify how to handle errors arising in the server stubs, or let the ILU runtime handle them in the default way: print an error message to stdout and quit the listener or connection worker. The server module may assert control over the association between *object-handles* and objects in the server by supplying an `ObjectTable`, or let the ILU runtime manage the association in its default way.

The server module continues by calling `Ilu.Export_Server`, specifying the protocol and transport combinations through which the server should be contactable. Due to internal restrictions in the current runtime, this procedure should be called exactly once.

Each true object should be a subtype of `Ilu.Object`; the implementor of the true object is responsible for ensuring that the `ilu_is_surrogate` is filled in with `FALSE` and that the `Ilu_Get_Server`, `Ilu_Get_Type`, and `ILU_Qua_Type` methods have reasonable behavior. The `ilu_is_surrogate` field defaults to `FALSE`, and the object type declared in a Modula-3 interface generated by the `m3-stubber` from an ISL interface takes care of implementing `Ilu_Get_Type`, so a programmer using the stubs needs to worry only about `Ilu_Get_Server` and `ILU_Qua_Type`.

Once a true object has been created, and `Ilu.Export_Server` has been called, the server can export individual objects. This can be done through a name service or by passing the object to another module among the arguments, results, or exception parameter contents of a call on a different language-specific object. The Simple Binding facility described later is integrated with ILU. To use a non-integrated name service, the object's string binding handle and most specific type ID are needed; they can be determined by calling `Ilu.SbhFromObject(`*obj*`)` and `Ilu.IdOfObjectType(`*obj*`.ILU_Get_Type())`.

6.4 ILU API for Modula-3

The full API is presented in the previous section.

ILU currently supports DEC SRC Modula-3 version 2.08 — which lacks finalization. When an application program — any combination of client and server modules — knows it is done with a particular object, it can explicitly free the resources associated with that object. This is done by invoking the `ILU_Close` method on that object.

It is always safe — but may be expensive — to invoke `ILU_Close` on a surrogate object or on a true object that will be found by the `HandleToObject` method of its server's `ObjectTable`. The `HandleToObject` method of the default `ObjectTable` implementation will not find a true object after `ILU_Close` has been called on that object.

6.4.1 Simple Binding

The Simple Binding functionality is available through the following interface.

```
INTERFACE IluSimpleBind;
FROM IluBasics IMPORT Failed;
IMPORT Ilu;

<*PRAGMA lL, Ll, Main*>

<*Main invariant holds*>

TYPE Cookie <: REFANY;

PROCEDURE Publish (obj: Ilu.Object): Cookie RAISES {Failed};

PROCEDURE Withdraw (obj: Ilu.Object; c: Cookie) RAISES {Failed};

PROCEDURE Lookup (iid: Ilu.InstanceId; ot: Ilu.ObjectType): Ilu.Object
  RAISES {Failed};

END IluSimpleBind.
```

The instance ID used in the `Lookup` call is what's called an *object ID* in chapter 1. It is the concatenation of: (1) the object handle, as determined by the server's `Ilu.ObjectTable`; (2) an at-sign (`@`); and (3) the server ID, determined in the call on `Ilu.InitTrueServer`.

6.5 Generating ILU stubs for Modula-3

To generate Modula-3 stubs from an ISL file, you use the program m3-stubber. Five files are generated from the '`.isl`' file:

- '*interface-name*`.i3`' contains the Modula-3 renderings of the types, exceptions, and constants declared in the interface, plus some items needed to import or export objects of types declared in the interface;
- '*interface-name*`_x.i3`' is a private interface between the following three implementation modules;
- '*interface-name*`_y.m3`' contains code useful to both server and client stubs;

- '*interface-name*`_c.m3`' contains the client stubs; and
- '*interface-name*`_s.m3`' contains the server stubs for the interface.

Typically, client and server programmers directly reference only the first of these five files.

```
% m3-stubber foo.isl
translating interface foo to ./foo.i3...
private interface for foo to ./foo_x.i3...
common code for interface foo to ./foo_y.m3...
client stubs for interface foo to ./foo_c.m3...
server stubs of interface foo to ./foo_s.m3...
%
```

6.6 Libraries and Linking

Clients of an ILU interface need to link with all but the server stubs; servers need to link with all five modules. It's convenient to make a library containing all five modules and let the linker worry about the details of which are needed; the `imake` macro `IluM3Files` (see later) conveniently generates the names of all five modules.

Both clients and servers also need to link with the libraries '*ILUHOME*`/lib/libilu-m3.a`' and '*ILUHOME*`/lib/libilu.a`' (in this order, as the former uses functions in the latter). Because the former library contains only Modula-3 code, and the latter only C code, invocations of the `m3` command need to mention the latter library only when a complete program is being built.

7 Using ILU with Python

7.1 Introduction

This document is for the Python programmer who wishes to use ILU. The following sections will show how ILU is mapped into Python constructs and how both Python clients and servers are generated and built.

7.2 The ISL Mapping to Python

7.2.1 Names

In general, ILU constructs Python symbols from ISL names by replacing hyphens with underscores. For example, an ISL object type `T-1` would correspond to the Python class `T_1`. Any place an ISL name appears as part or all of a Python identifier, this translation occurs.

7.2.2 Interface

Each ISL `interface` *I* generates two Python modules: one named *I* containing common definitions, and another named *I*`__skel` containing skeletons (server stubs). For example, `INTERFACE map-test;` generates the Python modules `map_test` and `map_test__skel`, contained in the files '`map_test.py`' and '`map_test__skel.py`', respectively.

7.2.3 Constant

ISL constants translate to Python variables initialized to the specified value. For example,

```
CONSTANT pi : real = 3.14159265358979323846;
```

maps to

```
pi = 3.14159265358979323846e0
```

7.2.4 Exception

An ISL exception translates to a Python variable initialized with a string representing the exception. These variables are used in Python `raise` statements in object implementation code, and in `try ... except` statements in client code. For example, the declaration

```
EXCEPTION division-by-zero;
```

in the interface `map-test` maps to the following statement in '`map_test.py`':

```
division = 'map-test: division-by-zero'
```

7.2.5 Types

7.2.5.1 Basic Types

The basic ISL types have the following mapping to Python types:

1. BYTE, BOOLEAN, SHORT CHARACTER, CHARACTER, SHORT INTEGER, INTEGER, and SHORT CARDINAL all map to Python int.
2. LONG INTEGER, CARDINAL, and LONG CARDINAL all map to Python long int.
3. SHORT REAL and REAL map to Python float.
4. LONG REAL maps to the Python type ilu_longreal, a type implemented by the ILU Python runtime. This type has limited functionality, but can be passed around without loss of precision, converted to float or int, and compared. A value of this type may be constructed by calling `ilu.LongReal()`.

7.2.5.2 Enumeration

The names of Python variables for enumeration values are formed by prepending the enumeration type name and two underscores ("__") to the enumeration value name. There is also a dictionary for each enumeration type that maps an enumeration value to a string corresponding to its Python enumeration value name. The name of the dictionary is "imageOf" followed by three underscores ("___") followed by the enumeration type name.

For example,

```
TYPE color = ENUMERATION red, dark-blue END;
```

maps to

```
color__red = 0
color__dark_blue = 1
imageOf___color = {
  color__red: 'color__red',
  color__dark_blue: 'color__dark_blue'}
```

7.2.5.3 Array

An ISL array maps into a Python list with the specified number of elements. Tuples as well as lists are accepted as input, but lists are always produced as output from ILU. For an array of short character or byte, strings are also accepted, in which case the `ord()` of each character in the string is sent. Be careful using these alternate forms on input, since they will go across as is for objects in the same address space, but will be changed to lists for remote objects.

7.2.5.4 Sequence

An ISL sequence of short character maps into a Python string.

All other ISL sequence types map into Python lists. Tuples as well as lists are accepted as input, but lists are always produced as output from ILU. For a sequence of byte, strings are also accepted, in which case the `ord()` of each character in the string is sent. Be careful using these alternate forms on input, since they will go across as is for objects in the same address space, but will be changed to lists for remote objects.

7.2.5.5 Record

ISL records map into Python dictionaries, with the field names as string keys, and the field values as the corresponding values.

For example, a record value of the ISL type

```
TYPE segment = RECORD left-limit : integer, right-limit : integer END;
```

with a left-limit of -3 and a right-limit of 7 would map to

```
{'left_limit': -3, 'right_limit': 7}
```

7.2.5.6 Union

An ISL union maps into a Python tuple with two components: an integer discriminator, and the discriminated value. There are three possibilities:

1. If the discriminator matches one of the union case values of an arm, the second component is of the type specified by that arm.
2. If the discriminator matches no union case values and there is a default arm, the second component is of the type specified by the default arm.
3. If the discriminator matches no union case values and there is no default arm but the union has the `OTHERS` attribute, the second component is `None`.

7.2.5.7 Object

Each ISL object type is mapped into a Python class. These classes have the methods specified in the ISL, as well as some built-ins.

7.2.5.8 Optional

A value corresponding to the ISL type `OPTIONAL` *T* may be `None` (indicating the null case) in addition to the values of the type *T*.

7.2.6 Methods and Parameters

ISL methods of an object type map to Python methods of the corresponding class. `IN` and `INOUT` parameters appear in the Python method signature in the same order as they do in ISL.

Let us define a *result* value to be either a return value (corresponding to a method's return type) or an `INOUT` or `OUT` parameter. Result values are returned by the Python method as a tuple, with the return value (if present) appearing before any parameters. If the method has only one result value, then it is simply returned (i.e., a tuple of length one is *not* constructed to hold this value). If the method has no result values, then `None` is returned.

7.3 Using an ILU module from Python

The ILU runtime interface is in the Python module `ilu`. Python definitions for ISL `INTERFACE` *I* are in the Python module *I*. As with any other modules in Python, these modules are imported using the `import` statement.

A client program may create an ILU object in one of three ways:

1. Knowing the string binding handle `sbh` and class `cl` of an object, call `ilu.ObjectOfSBH(cl, sbh)` which returns an instance of that class. For example, to obtain an instance of ISL type `square` from `INTERFACE shapes` whose string binding handle is `sbh`, one would call `ilu.ObjectOfSBH(shapes.square, sbh)`.
2. Knowing the object ID `(sid, ih)` and class `cl` of an object that has been published using the simple binding service, call `ilu.Lookup(sid, ih, cl)` which returns an instance of that class (or `None` if the lookup fails).
3. Receive an instance as a result value from a method call that returns an object type or has an object type as an `INOUT` or `OUT` parameter.

7.4 Implementing an ILU module in Python

A Python module that implements ILU objects of types defined in `INTERFACE` *I* also imports from *I*`__skel`. This gives access to the skeleton classes from which implementation classes inherit.

7.4.1 Implementation Inheritance

An implementation of object type *T* from interface *I* needs to inherit from the class *I*`__skel.`*T*. If there is inheritance in the ISL, and an implementation of a subtype wants to inherit from an implementation of a supertype, the skeleton class must be appear in the list of base types before the implementation class.

For example, objects for the ISL

```
INTERFACE j;
```

```
TYPE c1 = OBJECT METHODS one() END;
TYPE c2 = OBJECT METHODS two() END;
TYPE c3 = OBJECT SUPERTYPES c1, c2 END METHODS three() END;
```

could be implemented in Python by

```
import ilu, j, j__skel

class c1(j__skel.c1):
    def one(self):
        ...

class c2(j__skel.c2):
    def two(self):
        ...

class c3(j__skel.c3, c1, c2):
    def three(self):
        ...
```

In this case `c3`'s method `one` is implemented by `c1.one` and `c3`'s method `two` is implemented by `c2.two`.

7.4.2 True Servers

Each object exported by an implementation must belong to a true server, an instance of the Python type `ilu_Server` which is implemented by the ILU runtime. An `ilu_Server` can be created by calling the function `ilu.CreateServer(serverID = None, transport = None, protocol = None, objectTable = None)`, which returns a value of type `ilu_Server`. If *serverID* is a string, it specifies the server ID; if it is `None`, one will be invented automatically. The *transport* and *protocol* arguments are strings to choose a specific transport or protocol, or `None` to let them default. The *objectTable* argument allows specification of a callback function for creating true instances on demand. The callback function should take one argument, a string, which is the object ID of the instance to be created, and return a true instance.

The first time a true server is created, it becomes the default server. The default server is used for an exported object if a server is not otherwise specified. If an object is exported before any servers have been created, one will be created automatically using default parameters and a message to that effect will be written to stderr.

An object of type `ilu_Server` has a method `id()` that returns its server ID.

7.4.3 Exporting Objects

An object can be exported in one of three ways:

1. The object's string binding handle may be obtained by calling its method `IluSBH()` and communicating this somehow to a client, who then turns the handle back into an object by calling `ilu.ObjectOfSBH(cl, sbh)`.
2. The object may be published using the simple binding service by calling its method `IluPublish()`. In order for this to be effective, the object must have a well-known object ID, or the object ID must be communicated to clients, so clients can know what to pass to `ilu.Lookup`. The object ID is a function of the object's instance handle and its server's server ID.
3. The object may be returned by a method or passed back in a method's `INOUT` or `OUT` parameter.

An object's instance handle can be controlled by setting the instance variable `IluInstHandle` before the object is first exported. If this instance variable is not set, and instance handle will be invented automatically.

An object's server can be controlled by setting the instance or class variable `IluServer` to a value of type `ilu_Server`. The value of this variable at the time an object is first exported will be used as the server for that object. If such a variable is not set, the default server is used.

7.4.4 Animating Servers

Running the ILU main loop by calling `ilu.RunMainLoop()` brings the true servers to life. This function does not return until `ilu.ExitMainLoop()` is called. If you are using ILU with `Tkinter`, run the main loop by calling `ilu_tk.RunMainLoop()`, rather than using either the ILU or `Tkinter` main loops. `ilu_tk.RunMainLoop` sets things up so that both Tk and ILU events are handled.

7.4.5 Using Alarms

In order to schedule a Python function to be called at a certain time in the future when executing the ILU main loop, an `ilu_Alarm` may be used. Objects of this type are created by calling `ilu.CreateAlarm()`. An `ilu_Alarm` must be set to have any effect.

The alarm's method `set(time, proc, args)` is used to set the alarm. The `int`, `float`, or `ilu_FineTime` `time` argument is the time at which the alarm will fire; the `proc` argument is the Python function that will be called when the alarm fires; and the `args` argument is a tuple of arguments to be passed to `proc`. The tuple `args` must match `proc`'s signature. For example, if `proc` is declared `def P(a, b):` then `args` must be a two-tuple. Likewise, if `proc` takes only one argument then `args` must be a one-tuple, or if no arguments then a zero-tuple.

The function `ilu.FineTime_Now()` may be called to obtain ILU's idea of the current time. A value `sec` of type `int` or `float` in units of seconds may be converted to type `ilu_FineTime` by calling `ilu.FineTime(sec)`. Values of type `ilu_FineTime` may be compared, added, and subtracted. These operations may be used to construct values representing any relative time (subject to precision and range limitations), which is what is needed by an alarm's `set` method.

The alarm may be set multiple times with different arguments, in which case the parameters of the most recent call to `set` are in effect. Thus, once an alarm fires, it may be reused by calling `set` again.

An alarm may be unset by calling its method `unset()`.

7.5 Using the Simple Binding Service

An object may be published using the simple binding service by calling its method `IluPublish()`. An object may be unpublished by calling its method `IluWithdraw()`.

A published ILU object may be obtained by calling `ilu.Lookup(sid, ih, cl)`, where `sid` is object's server's server ID, `ih` is the object's instance handle, and `cl` is its class.

7.6 Summary of the ILU Python Runtime

Exported from module `ilu`:

- `def CreateAlarm():`

  Creates an object of type `ilu_Alarm`.

- `def CreateServer(serverID = None, transport = None, protocol = None):`

  Used to create an `ilu_Server` object with the specified serverID, transport, and protocol. If *serverID* is `None`, an identifier will be invented automatically. If transport or protocol are `None`, they will default to `'sunrpcrm|tcp_localhost_0'` and `'sunrpc_'`, respectively. The first time `CreateServer` is called, the server so created becomes the default server. If there is no default server when one is required, one will be created using default parameters and a message will be issued on stderr.

  An `ilu_Server` object has an `id` method which returns the string identifier of that server.

- `def DefaultServer():`

  Returns the default server.

- `def ExitMainLoop():`

  Exits the ILU main loop, assuming it is running.

- `FALSE = 0`

- `def FineTime(sec):`

  Converts its `int` or `float` argument *sec* in units of seconds to type `ilu_FineTime`. Objects of this type can be compared, added, subtracted, and converted to `int` or `float`. The main use of objects of this type is in setting alarms.

- `FineTimeRate = ...`

  The precision of type `ilu_FineTime` in seconds is the reciprocal of this constant.

- `def FineTime_Now():`

  Returns the current time as an `ilu_FineTime` object.

- `def FormSBH(objectID, contactInfo):`

  Returns the string binding handle corresponding to the object id *objectID* and contact info *contactInfo*. This is the inverse of `ParseSBH`.

- `IluGeneralError`, `IluProtocolError`, and `IluUnimplementedMethodError` are all strings that may occur as exceptions from the ILU runtime.

- `def LongReal(v):`

  Converts its `int`, `float`, or sixteen-integer `list` or `tuple` argument to type `ilu_LongReal`. In case of a list or tuple, the elements encode the bytes of the IEEE long real value, from most significant to least.

- `def LookupObject(sid, ih, cl):`

  Returns the object with object server ID *sid*, object instance handle *ih*, and Python class `cl`, assuming it was previously published using the simple binding service. If the lookup fails, `None` is returned.

- `def ObjectOfSBH(cl, sbh):`

  Returns the object corresponding to the Python class *cl* and string binding handle *sbh*.

- `def ParseSBH(sbh):`

  Returns the pair (object id, contact info) corresponding to the string binding handle *sbh*.

- `def RegisterInputHandler (file, handler_fn):`

  Sets up an association between the *file* (which must be a file object opened for reading), and the *handler_fn* (which must be a callable function with no arguments) so that *handler_fn* is called whenever input is available on *file*. This is useful for implementing a server that also responds to commands typed to its standard input, for example. Passing a value of `None` for the *handler_fn* removes the association.

- `def RunMainLoop():`

  Runs the ILU main loop.

- `def SetMainLoop(DoEvent, RegisterInput, CancelInput, RegisterOutput, CancelOutput, CreateAlarm, SetAlarm, CancelAlarm)`

  The purpose of this function is to be able to use a foreign main loop (such as for a user interface toolkit) with an ILU server. The details will not be described here. Look at the runtime module `ilu_tk` for an example of its use.

- `def SetDebugLevel(bits):` or `def SetDebugLevel(string):`

  Sets the ILU kernel debugging flags according to its `int` argument, if an int is specified, or via the colon-separated list of debug switches, if a string is specified. See the Debugging section of the ILU Manual for more information on these switches.

- `TRUE = 1`

- `def TypeID(cl):`

  Returns the ILU unique type identifier corresponding to the Python class *cl*.

- `def TypeName(cl):`

  Returns the ILU type name corresponding to the Python class *cl*.
- `Version = ...` is the ILU version string.

Built-in methods of ILU objects:

- `IluObjectID()` returns the object ID of the object.
- `IluPublish()` publishes the object using the simple binding service.
- `IluSBH()` returns the object's string binding handle.
- `IluTypeID()` returns the unique type identifier of the object's ILU type.
- `IluTypeName()` returns the type name of the object's ILU type.
- `IluWithdraw()` undoes the effect of IluPublish().

Special attributes of ILU true objects: One or more of the following attributes may be set in a true (implementation) object of an ISL object type to control certain aspects of that object.

- `IluInstHandle`, a string instance variable, gives the object's instance handle. If not present, an instance handle is invented automatically.
- `IluServer`, a variable of type `ilu_Server`, determines the object's server. This can be an instance or a class variable. If not present, the default server is used.

7.7 Stub Generation

To generate Python stubs from an ISL file, use the program `python-stubber`. Two files are generated from each ISL `INTERFACE` *name*:

- '*name*`.py`' containing code for constants, exceptions, and types defined in the interface, and
- '*name*`__skel.py`' containing code for the skeletons (server stubs) for object types defined in the interface.

8 Protocols and Transports

When two modules of a program are in different address spaces, or use different data representations, ILU forms messages to send across the inter-module boundary; we call a particular way of forming and interpreting these messages an *RPC protocol* (sometimes simply *protocol*). These messages may be transported between address spaces in different ways; we call a particular way of moving messages a *transport*. This chapter describes the various kinds of available ILU protocols and transports. ILU is extensible: additional RPC protocols and transports can be added, either at compile-time or run-time; this chapter does not describe how to do so.

When an ILU kernel server exports objects, it does so via one or more contact stacks. Each stack has an RPC protocol at the top of the stack, forming and interpreting messages, and one or more layers of transport below the protocol layer, transforming or communicating the messages in various ways. A contact stack is specified by a protocol-info string and a sequence of transport-info strings; the syntax of these strings is defined in this chapter.

8.1 Protocols

8.1.1 The Abstract ILU Message Protocol

Before describing any particular protocol, we will describe the abstract ILU protocol, which is layered on top of each actual protocol. It is quite simple. Two types of messages are used, one to communicate parameters to a true method, and the other to communicate results and/or exceptions from the true method to surrogate caller. Parameters and values are encoded according to a simple abstract external data representation format. This abstract protocol identifies what information is passed between modules without specifying its exact mapping to bit patterns.

8.1.1.1 Messages

The first type of message is called a *request*. Each request consists of a code identifying the method being requested, an authentication block identifying the principal making the call, and a list of parameter inputs to the method being called. The method is identified by passing the one-based ordinal value (that is, the index of the method in the list of methods, beginning with one) of the method, in the list of methods as specified in the ISL description of the class which actually defines the method. No more than 65278 (1-0xFEFF) methods may be directly specified for any type (though more methods may be inherited by a type). Method codes 0xFF00 to 0xFFFF are reserved for ILU internal use. The principal is identified by a block of authentication credentials information which varies depending on the specific authentication protocol used. These credentials may be either in the request header, or may appear as a parameter of the request. (Note: There should also be an ILU protocol version number somewhere here, but there isn't (yet).)

The *result* message is used to convey return values and exception values from the true method back to the caller. It consists of a Boolean value, indicating whether the call was successful (for `TRUE`) or signalled an exception (for `FALSE`). If successful, the return value (if any), follows, followed by the values of any `Out` parameters, in the order they are specified as parameters. If an exception was signalled, a value between 1 and 2^16-1 follows, indicating the ordinal value specific exception in the list specified in the definition of the method, followed by a value of the exception type, if any was specified for the exception.

8.1.1.2 Parameter Types

Simple numerical values, of types `integer`, `cardinal`, `real`, or `byte`, are passed directly.

`Character` values are passed as integer values in the range [0,2^16-1]. `Short character` values are passed as integer values in the range [0,2^16-1]. `Long character` values are passed as integer values in the range [0,2^32-1].

`Enumeration` values are passed as integer values in the range [0,2^16-1], the value being the zero-based ordinal value of the corresponding enumeration value in the original list of enumeration values in the definition of the enumerated type.

`Boolean` values are passed as as integer values of either 0, for `FALSE`, or 1, for `TRUE`.

`Optional` values are passed by first passing a Boolean value, with `TRUE` indicating that a non-`NIL` value is being passed, and then only in the non-`NIL` case passing a value of the optional value's indicated type.

`Sequence` values are passed by first passing a count, as an integer in the range [0,2^32-1] for sequences without limits, or for sequences with limits greater than 2^16-1, or an integer in the range [0,2^16-1], for sequences with limits less than 2^16, indicating the number of elements in the sequence, and then that number of values of the sequence's base type.

`Array` values are passed by passing a number of elements of the array's base type corresponding to the size of the array.

`Record` values are passed by passing values of types corresponding to the fields of the record, following the order in which the fields are defined in the ISL definition of the record.

`Union` values are passed by passing a value of the discriminant type, which indicates which branch of the union constitutes the union's actual type, usually followed by a value of the union's actual type. If the discriminant value indicates a branch of the union which has no associate value, only the discriminant value is passed.

`Object` values are passed in several different forms, depending on whether or not the object value is in the discriminator position, whether or not the object's type is a `singleton` type, and whether or not the object reference is `NIL`.

1. The first form is used when the object is in the discriminator position (that is, is the instance upon which the method is being invoked), and is an instance of a `singleton` type. In this case, the object is already known to both sides, and the object is passed implicitly; that is, no actual bytes are transmitted.
2. The second form is used when the object is in the discriminator position, but is not of a `singleton` type. In this case, the instance handle and server ID of the object are passed as `sequence of short character` values. Both the instance handle and server ID must be passed, as the true object previously at the "known address" for the object may have been replaced by a different object with the same instance handle, in a different kernel server.
3. In the third case, the object is being passed as a normal parameter, that is, not in the discriminator position. In this case, the full string binding handle of the object is passed as a `sequence of short character` value.
4. If the object being passed as a normal parameter is the CORBA Nil object reference, it is passed as the `sequence of short character` value of length zero.

8.1.2 The ONC RPC Protocol

This section describes the mapping of the abstract ILU protocol into the specific on-the-wire protocol used with ONC RPC[1] One of the major goals of this mapping is to preserve compatibility with existing Sun RPC services that can be described in ISL.

A protocol info string for ONC RPC has the form `sunrpc_2_`*program-number*`_`*program-version* where *program-number* and *program-version* may be specified either in decimal, or in hexidecimal with a leading string of `0x`. The program number for non-native-ONC-RPC ILU object types is always (in ILU 2.0) `0x61A78`, and the program version varies depending on the specific object type.

Use of ONC RPC requires use of a boundaried transport below it.

8.1.2.1 Message Mappings

The request message used is that specified by the ONC RPC protocol. The ILU method index is encoded as a 32-bit number in the "proc" field in the ONC RPC request header. Principal identification is passed in the "cred" field of the ONC RPC request header. By default, ILU will pass the `AUTH_UNIX` authentication information, if no authentication method is specified for the method. For non-singleton object types, the ONC RPC program number passed in the "prog" slot is always 0x00061a78 (for ILU 2.0) and the version number passed in the "vers" slot is the CRC-32 hashed value of the MSTID for the object type on which the method being invoked is defined. For singleton classes, the program number and version specified in the singleton information is used.

[1] RPC: Remote Procedure Call Protocol Specification, Version 2; R. Srinivasan. IETF RFC 1831, August 1995.

The "mtype" field is set to `CALL`. The indicated "rpcvers" is 2. A monotonically increasing 32-bit serial number is used in the "xid" field.

The reply message used is that specified by the ONC RPC protocol. The "mtype" field is set to `REPLY`. The "stat" field is always set to `MSG_ACCEPTED`. In the `accepted_reply`, the authentication verifier is always NULL. The "stat" field may be non-zero, to signal one of a small number of "standard" exceptions, or may be zero. This header is then followed by one of three forms: If a "standard" exception was raised, nothing. If the method has no exceptions, the return values and out parameters (if any). If the method has any exceptions defined, a 32-bit value which specifies either successful completion (a value of 0), or an exception (a value greater than 0, which is the ordinal value of the particular exception being signalled in the list of exceptions specified for this method), followed by either the return value and out parameters (if any), in the case of successful completion, or the exception value (if any), in the case of an exception.

8.1.2.2 Mapping of Standard Types

The mapping of ILU types into ONC RPC types is accomplished primarily by using the appropriate XDR[2] representation for that type.

`Short integer` and `integer` types are represented with the XDR Integer type. `Long integer` types are represented as an XDR Hyper Integer.

`Short cardinal`, `byte`, and `cardinal` types are represented with the XDR Unsigned Integer type. `Long cardinal` types are represented as an XDR Unsigned Hyper Integer.

`Short real` numbers are encoded as XDR Floating-point. `Real` numbers are encoded as XDR Double-precision Floating-point. `Long real` numbers are encoded as XDR Fixed-length Opaque data of length 16.

`Array` values are encoded as XDR Fixed-length Array, except for two special cases. If the array is multi-dimensional, it is encoded as a flat rendering into a single-dimensional array in row-major order (the last specified index varying most rapidly). If the array is of element-type `byte` or `short character`, it is encoded as an array of one (in the one-dimensional case) or more (in the greater-than-one dimensional case) values of XDR Opaque Data.

`Record` values are encoded as XDR Structures.

`Union` values are encoded as XDR Discriminated Unions, with a discriminant of type "unsigned int" containing the ILU `short cardinal` discriminant.

`Enumeration` values are encoded as XDR Unsigned Integer (note that this is different from XDR Enumerations, which are encoded as XDR Integer).

---

[2] XDR: External Data Representation Standard; R. Srinivasan. IETF RFC 1832, August 1995.

`Boolean` values are encoded as XDR Unsigned Integer, using the value 0 for `FALSE` and the value 1 for `TRUE`.

`Sequence` values are encoded as XDR Variable-length Arrays, except for several special cases. Sequences of `short character` are encoded as XDR String, sequences of `byte` are encoded as XDR Variable-length Opaque Data, and sequences of `character` are encoded as XDR String, where the string is the UTF-2 encoding of the Unicode characters in the sequence.

`Optional` values are encoded as an XDR Boolean value, followed by another encoded value, if the Boolean value is TRUE.

Instances of an `object` type are encoded as either zero (in the case of a method discriminant of a singleton type), or one, values of type XDR String.

8.1.3 The Xerox Courier Protocol

This section describes the mapping of the abstract ILU protocol into the specific on-the-wire protocol used with Xerox Courier[3]. One of the major goals of this mapping is to preserve compatibility with existing Xerox Courier services that can be described in ISL. Unfortunately, many if not most important Courier services use *bulk data transfer*, something that is still only planned for ILU.

A protocol info string for Xerox Courier has the form `courier_`*program-number*`_`*program-version* where *program-number* and *program-version* may be specified either in decimal, or in hexidecimal with a leading string of `0x`. The program number for non-singleton ILU object types is always (in ILU 2.0) `0x0010xxxx`; the specific program number and the program version varies depending on the specific object type.

Use of Xerox Courier requires use of a boundaried transport below it.

8.1.3.1 Message Mappings – Courier Layer 3

The request message used is the `CallMessageBody` specified in section 4.3.1 of the Courier protocol. A monotonically increasing 16-bit serial number is passed in the `transactionID` field; a 32-bit program number is passed in the `programNumber` field, a 16-bit number is passed in the `versionNumber` field; the ILU method index is passed as a 16-bit value in the `procedureValue` field. The program number is calculated by computing the CRC-32 hash value of the MSTID of the object type on which the method is defined, then forming a program number by using the value `0x0010` for the high-order 16 bits, and the high-order 16 bits of the CRC for the low-order 16 bits of the program number. The version number is the low-order 16 bits of the CRC.

[3] Courier: The Remote Procedure Call Protocol; Xerox Corporation, XNSS 038112, 1981

Successful replies are sent using the Courier `ReturnMessageBody` specified in section 4.3.3 of the Courier specification. The `procedureResults` field contains the return value, if any, followed by the `INOUT` and `OUT` parameter values, if any.

User exceptions are signalled using the `AbortMessageBody` specified in section 4.3.4 of the Courier specification. The `errorValue` field contains a value greater than 0, which is the ordinal value of the particular exception being signalled in the list of exceptions specified for this method. The `errorArguments` field contain the exception value, if any.

System exceptions (of exception type `ilu.ProtocolError`) are signalled using the `RejectMessageBody` message of section 4.3.2. The `rejectionDetail` field of the message contains the `ProtocolError` detail.

8.1.3.2 Mapping of Standard Types – Courier Layer 2

The mapping of ILU types into Courier types is accomplished primarily by using the appropriate Courier Layer 2 representation for that type.

`Short integer` and `integer` types are represented with the Courier `Integer` and `Long Integer` types. `Long integer` types are represented as an `integer` followed by a `cardinal`.

`Short cardinal`, `byte`, and `cardinal` types are represented with the Courier `cardinal`, `cardinal`, and `long cardinal` types, respectively. `Long cardinal` types are represented as a big-endian (most significant 16 bits first) Courier array of 4 `cardinals`.

As the Courier protocol does not have any mapping for floating point values, `short real` numbers are passed as a Courier `long cardinal`, `real` numbers are encoded as a big-endian array of two Courier `long cardinal` values, and `long real` numbers are encoded as big-endian array of four Courier `long cardinal` values.

`Array` values are encoded as Courier one-dimensional `arrays`. If the array is multi-dimensional, it is encoded as a flat rendering into a single-dimensional array in row-major order (the last specified index varying most rapidly). If the array is of type `byte` or `short character`, the contents of the ILU value are packed into a Courier array of `unspecified` two values per array element, so that the Courier array is half the length of the actual ILU array.

`Record` values are encoded as Courier `record` values.

`Union` values of union types whose discriminant type can be mapped to a 16-bit value type in the range [0,2^16-1] are passed as Courier `choice` values. Other unions are passed as a Courier `long cardinal`, followed by the value of the union's indicated type (if any).

`Enumeration` values are encoded as Courier `enumeration` values.

`Boolean` values are encoded as Courier `boolean` values.

`Sequence` values are encoded as Courier `sequences`, except for several special cases. Sequences of N `short characters` or `bytes` are encoded as either a Courier `cardinal`, for sequences with limits less than 2^16, or `long cardinal`, for sequences with no limits or limits greater than 2^16-1, value of *N*, followed by `(`*N*`+1)/2` values of Courier `unspecified`, each such value containing two `short character` or `byte` values, packed in big-endian order.

`Optional` values are encoded as an Courier `boolean` value, followed by another encoded value, if the Boolean value is `TRUE`.

Instances of an `object` type are encoded as either zero (in the case of a method discriminant of a singleton type), or one values of ISL `short sequence of short character`. CORBA Nil object references are represented as a zero-length `short sequence of short character`.

8.1.4 The OMG Internet Inter-Orb Protocol (IIOP)

A protocol info string for IIOP has the form `IIOP_1_0`.

Use of IIOP requires use of a reliable unboundaried transport below it.

8.2 Transports

A transport stack consists of a sequence of transport layers. The last, or "bottom", layer does some kind of low-level I/O; the other layers are "filters" or "modifiers" on the transport services provided by the lower part of the stack. That is, every tail of a transport stack implements an abstraction called simply "a transport" (in English; in C, it is `ilu_Transport`); each transport layer (except the bottom) implements a transport in terms of another transport (the one implemented by the rest of the stack).

A transport stack is specified by a sequence of strings, each one of which specifies a transport layer. This section is a catalog of the built-in kinds of transport layers and their specification strings.

Some transports convey delimited messages (where each message is a byte sequence), others simply convey a byte sequence (that must be parsed into messages by something else). The former are called boundaried, the latter are not. Some RPC protocols require a boundaried transport, others require non-boundaried transports.

Some transports are reliable, and some aren't. Unreliable transports are deprecated in ILU, but included for interoperability with existing software that uses only unreliable transports. With these transport, messages may be delivered more than once. The ILU implementation of UDP on the server side filters out multiple receipts of the same request. Asynchronous methods may not be called over this transport mechanism, as reliable delivery of the request packet cannot be recognized by the client side. Non-asynchronous methods use the reply message as an acknowledgement that the request was received.[4]

8.2.1 TCP

A `TCP` transport layer is reliable, not boundaried, and goes on the bottom.

A `TCP` transport is specified by a *transport-info* of the form `tcp_`*host*`_`*port*.

The *host* needs to convey an IP address. The *host* can either be a dotted decimal notation of an IP address (e.g., `13.2.116.14`), or be a hostname that can be mapped into an IP address. Also, the *host* in a `TCP` *transport-info* used to create a mooring (i.e., one used to tell a server how to export itself) can use special notations to mean "any IP address of this host". Those notations are `0`, `0.0.0.0`, and `localhost`. These special notations are replaced with either a name or address of the host when an SBH is produced; an SBH cannot contain such special notations. `localhost` is replaced by a name; `0` and `0.0.0.0` are replaced by an address. It is usually difficult if not impossible to unambiguously determine a name; the form `0` should be used instead of `localhost` whereever possible.

The *port* can either be a decimal string identifying a specific port (which, of course, must not be used for anything else), or, for a mooring, be `0`, which constitutes a request for a new, unused port. In the latter case, a decimal string for that port will be substituted in the transport-info.

Examples are:

```
tcp_augustus_0
tcp_13.2.116.14_12321
tcp_localhost_12321
tcp_0_0
```

8.2.2 UDP

A `UDP` transport is boundaried, not reliable, and on the bottom.

A `UDP` transport is specified by a *transport-info* of the form `udp_`*host*`_`*port*. The *host* and *port* convey IP host address and port just as for `TCP`.

Examples are:

```
udp_augustus_0
udp_13.2.116.14_12321
udp_localhost_12321
udp_0_0
```

---

[4] Query: can requests be larger than the UDP packet size? [No] How then are they segmented? [They're not; replies aren't either] Note: This should probably be replaced by a reliable UDP protocol, in which each message is acknowledged by the receiver. This would allow use of asynchronous methods over UDP. Of course, ONC RPC would not cooperate.

8.2.3 SunRPC Record Marking

A `SunRPC Record Marking` transport is reliable and boundaried, and is built on top of some other transport that is reliable and not boundaried. A `SunRPC Record Marking` transport layer is specified by a *transport-info* of the form `sunrpcrm`.

Examples are:

```
sunrpcrm
```

8.2.4 In-Memory

`In-Memory` transport layers are not normally used or seen by users or applications; they are automatically created and used by ILU for cross-representation calls within an address space.

An `In-Memory` transport layer is reliable, boundaried, and on the bottom. An `In-Memory` transport layer is specified by the string `inmem`.

Example is:

```
inmem
```

9 Threads and Event Loops

9.1 Introduction

ILU can be used in either the single-threaded or the multi-threaded programming style. This chapter describes how.

The issue of *threadedness* appears at two levels: within a program instance, and again for an entire distributed system. We will first discuss the program level, and then the system level.

ILU factors its runtime support into a common kernel and several independent language-specific veneers; you will see this structure when you try to do certain non-vanilla things. The interface to the runtime kernel is '*ILUHOME*`/include/iluxport.h`'.

9.2 Multi-Threaded Programs

Some programming languages are defined to support multiple threads of control. Modula-3 is an example. Other language definitions are single-threaded, or are silent on this issue. Some of these, such as C and C++, can be used to write mutli-threaded programs with the use of certain libraries, coding practices, and compilation switches. ILU can be used in multi-threaded programs in both inherently multi-threaded languages and some of those where multi-threaded is an option.

ILU's runtimes for both Franz Common Lisp and Modula-3 support multi-threading; programmers do not need to do anything special in these languages.

ILU's runtimes for C++ and TCL assume single-threaded programming, and simply do not support multi-threading. For C++, this is a deficiency we hope to remedy soon. How wedded to single-threadedness is TCL?

ILU's runtime for C supports both single-threaded and multi-threaded programming; it assumes single-threading by default, and can be switched to multi-threading by a procedure call during initialization (described below).

What about Python?

ILU's runtime kernel defaults to supporting single-threaded operation, and can be switched to multi-theading by procedure calls during initialization. It is the responsibility of the language runtime to make these calls, if the language is inherently multi-threaded, or to offer the option of making these calls, if the language is optionally multi-threaded. A later subsection describes how to switch the kernel.

9.2.1 Multi-Threaded Programming in C

To switch the ILU ANSI C runtime from its default assumption of single-threadedness to multi-threaded operation, call `ILU_C_SetFork` (described in '*ILUHOME*`/include/iluchdrs.h`') before calling `ILU_C_Run`, `ILU_C_InitializeServer`, or anything that relies on a default `ilu_Server` existing. `ILU_C_SetFork` makes a feeble attempt to detect being called too late, returning a logical value indicating whether an error was detected (when an error is detected, the switch is not made). This detection is not reliable — the caller should take responsibility for getting this right.

Pass to `ILU_C_SetFork` a procedure for forking a new thread. This forking procedure is given two arguments: a procedure of one pointer (`void *`) argument and a pointer value; the forked thread should invoke that procedure on that value, terminating when the procedure returns.

9.2.2 Switching the Runtime Kernel to Multi-Threaded Operation

The kernel assumes single-threaded operation, and can be switched to multi-threading. To do so, three procedure calls must be made early in the initialization sequence, on `ilu_SetWaitTech`, `ilu_SetMainLoop`, and `ilu_SetLockTech`. See '`iluxport.h`' for details, and the Modula-3 and Common Lisp language-specific veneers (found in '*ILUSRC*`/runtime/m3/`' and '*ILUSRC*`/runtime/lisp/`') for usage examples.

9.3 Single-Threaded Programs

Users of ILU in single-threaded programs typically need to worry about only one thing: the main loop. To animate ILU server modules, a single-threaded program needs to be running the ILU main loop. This can be done, e.g., by calling `ILU_C_Run()` in C or `iluServer::Run` in C++. ILU also runs its main loop while waiting for I/O involved in RPC (so that incoming calls may be serviced while waiting for a reply to an outgoing call; for more on this, see the section on "Threadedness in Distributed Systems").

The problem is, many other subsystems also have or need their own main loop. Windowing toolkits are a prime example. When a programmer wants to create a single-threaded program that uses both ILU and another *main looped* subsystem, one main loop must be made to serve both (or all) subsystems. From ILU's point of view, there are two approaches doing this: (1) use ILU's default main loop, or (2) use some *external* (to ILU) main loop (this might be the main loop of some other subsystem, or a main loop synthesized specifically for the program at hand). ILU supports both approaches. Actually, ILU's runtime kernel supports both approaches. Currently no language veneers mention it. This is, in part, because it has no interaction with the jobs of the language veneers — application code can call this part of the kernel directly (from any language that supports calling C code).

9.3.1 ILU Main Loop Functional Spec

ILU needs a main loop that repeatedly waits for I/O being enabled on file descriptors (a UNIX term) and/or certain times arriving, and invokes given procedures when the awaited events happen. (Receipt of certain UNIX signals should probably be added to the kinds of things that can be awaited.) The main loop can be recursively invoked by these given procedures, and thus particular instances of the main loop can be caused to terminate as soon as the currently executing given procedure returns. This functionality can be accessed via the procedures `ilu_RunMainLoop` through `ilu_UnsetAlarm` in '`iluxport.h`'; these procedures are shims that call the actual procedures of whatever main loop is really being used.

9.3.2 Using ILU's Default Main Loop

In this approach, ILU's default main loop is made to serve the needs of both ILU and the other main-loop-using parts of the program. When the other main-loop-using parts of the program need to wait for I/O being enabled or a particular time arriving, you arrange to call the appropriate registration procedures (via, e.g., `ilu_RegisterInputSource`, `ilu_RegisterOutputSource`, `ilu_SetAlarm`) of the ILU main loop.

9.3.3 Using an External Main Loop

In this approach, you use an external (to ILU) main loop to serve the needs of ILU (as well as other parts of your program). This involves getting ILU to reveal to you its needs for waiting on I/O and time passage, and your arranging to satisfy these needs using the services of the external main loop. You do this by calling `ilu_SetMainLoop` early in the initialization sequence, passing a `ilu_MainLoop` metaobject of your creation. ILU reveals its needs to you by calls on the methods of this metaobject, and you satisfy them in your implementations of these methods.

Note that an `ilu_MainLoop` is responsible for managing multiple alarms. Some external main loops may directly support only one alarm. Later in '`iluxport.h`' you will find a general alarm multiplexing facility, which may come in handy in such situations.

See the files in '*ILUSRC*`/runtime/mainloop/`' for several examples of this approach (for the X Window System's various toolkits, like Motif, Xaw, XView, and Tk).

9.3.4 A Hybrid Aproach

Both of the above approaches rely on there being a certain amount of harmony between the functional requirements made by some main-looped subsystems and the functional capabilities offered by others. It also relies on the subsystems whose "normal" main loops are not used being open enough that you can determine their main loop needs. The conditions cannot be guaranteed in general. We've tried to minimize the main loop requirements of ILU, and maximize its openness.

We know of an example where neither of the above approaches is workable, and have a solution that may be of interest. See '*ILUSRC*`/etc/xview/`' for the (untested) code.

The problem is with the Xview toolkit (for the X Window System). Its main loop cannot be recursively invoked (a requirement of ILU), and the Xview toolkit is not open enough to enable use of any other main loop.

Our solution is to use Xview's main loop as the *top level* main loop, letting ILU use its own main loop when waiting on RPC I/O. Like the external main loop approach, this requires getting ILU to reveal its needs for waiting on I/O and time; unlike the external main loop approach, this requires *not* calling `ilu_SetMainLoop`. Instead of calling `ilu_SetMainLoop`, you call `ilu_AddRegisterersToDefault`, which causes ILU's default main loop to reveal ILU's needs to you — in addition to doing everything the default main loop normally does. (Actually, the multiple alarms of ILU have been multiplexed into one here for your convenience.) You register these needs with the Xview main loop, and run it at the top level.

This solution is not as good as we'd like; it does not provide a truly integrated main loop. Xview prevents us from doing much better.

9.4 Threadedness in Distributed Systems

In a distributed system of interacting program instances, you can (in principle, even if not (easily) in practice) trace a thread of control across remote procedure calls. Thus a distributed system, when viewed as a whole, can be seen to be programmed in either a single-threaded or multi-threaded style. ILU aims to minimize the consequences of the choice between in-memory and RPC binding, and this requires things not usually offered by other RPC systems. Some of these things are required by both the single-threaded and multi-threaded styles of programming distributed systems, for related but not quite identical reasons.

Forget RPC for a moment, and consider a single-threaded program instance. Method `m1` of object `o1` (we'll write this as `o1.m1`) may call `o2.m2`, which may call `o3.m3`, which may in turn call `o1.m1` again, which could then call `o3.m4`, and then everything could return (in LIFO order, of course). Late in this scenario, the call stack of the one thread includes two activations of the very same method of the same object (`o1.m1`), and another two activations of different methods of a common object (`o3`). All this is irrespective of module boundaries.

We want to be able to do the same thing in a distributed setting, where, e.g., each true object is in a different program instance. This means that while the ILU runtime is waiting for the reply of an RPC, it must be willing to service incoming calls. This is why ILU requires a recursive main loop in single-threaded programs.

In fact, one rarely wants single-threaded distributed systems. Indeed, the opportunities for concurrency are one of the main attractions of distributed systems. In particular, people often try to build multi-threaded distributed systems out of single-threaded program instances. While we hope this confused approach will fade as multi-threading support becomes more widespread, we recognize that it is currently an important customer requirement. Making single-threaded ILU willing to recursively invoke its main loop also makes single-threaded program instances more useful in a multi-threaded distributed system (but what you really want are multi-threaded program instances).

Threading is also an issue in RPC protocols. Some allow at most one outstanding call per connection. When using one of these, ILU is willing to use multiple parallel RPC connections, because they're needed to make nested calls on the same server.

10 ILU Customization

10.1 Introduction

ILU includes a number of internal interfaces that allow various functionality of the ILU kernel library to be replaced by user functionality.

- In a single-threaded application, the entire "main loop" can be replaced; in a threaded application, features of the thread system can be described to the ILU kernel so that the kernel will use them.
- New messaging protocols and inter-address-space transport mechanisms can be registered for use by an application.
- Object incarnation routines can be registered for 'kernel servers', so that true instances can be created on reference, rather than before reference.
- Handler routines can be registered to deal with `malloc()` failures in application-specific ways.
- Handler routines can be registered to perform application-specific actions when errors are signalled.
- If debugging is enabled at configuration time, specific sets of debugging messages can be selected at run time, and an application routine can be defined to intercept and handle debugging messages.

Most of these interfaces are defined in two ILU source files, '*ILUSRC*`/runtime/kernel/iluxport.h`' and '*ILUSRC*`/runtime/kernel/iluntrnl.h`'. We will not attempt to duplicate that documentation here, to avoid the inevitable errors when documentation is provided in two different forms; rather, this section of the manual will provide sketches of the interfaces, and refer the reader to the appropriate header files. For any discrepancies noted between the material here, and the material in the header files, the header files should be assumed to the 'truth'.

10.2 Event Loops and Threads

Every ILU address space uses either real threads, or some sort of event dispatching loop to simulate threads. ILU is thread-safe internally, and by default will continue to check its usage for even when event dispatching is used. Since many different thread systems and event dispatching loops exist, ILU provides interfaces to allow the user to describe the particular one that they're using to the ILU kernel. See section Section 9.4 [Threads and Event Loops], page 98 for more information on these topics. See '*ILUSRC*`/runtime/kernel/iluxport.h`' for documentation of the interfaces.

10.3 RPC Protocols and Data Transport Mechanisms

The ILU remote procedure call mechanism operates in layers.

- The application, at the highest layer, calls down into language-specific, interface-specific stub code, passing language-specific, method-specific values;
- that stub code calls down into an interface-independent language-specific runtime layer, passing method-independent, but language-specific values;
- that LSR calls down into language-independent, interface-independent ILU 'kernel' code, passing simple C values;
- the kernel calls into the code for a specific RPC protocol, such as ONC RPC, passing C values;
- the protocol code calls down into transport-mechanism code, passing 'messages' as opaque byte sequences;
- the transport code, which may consist of several layers, may perform various transformations on the messages, and finally conveys them to the peer transport layers.

ILU includes registration mechanisms to allow applications to add to the kinds of RPC protocols and data transport mechanisms that can be used.

10.3.1 RPC Protocols

Each `ilu_Protocol` object reflects a particular mapping of the abstract ILU RPC protocol onto a specific externally-defined RPC protocol. (See Section 8.2.4 [Protocols and Transports], page 94 for a discussion of the abstract ILU protocol, and how it is mapped to the ONC RPC protocol, and to the XNS Courier protocol.)

New RPC message protocols can be added to the ILU kernel by writing a new `ilu_Protocol` object, and calling the ILU kernel function `ilu_RegisterProtocol()`. register it. The structure and requirements of an `ilu_Protocol` object are defined in the file '*ILUSRC*`/runtime/kernel/iluntrnl.h`'; the methods of the protocol are considered to be 'inside' the ILU kernel, and must therefore conform to all ILU locking and error conventions. The locking conventions are discussed in '*ILUSRC*`/runtime/kernel/iluxport.h`'; the error conventions are documented in

'*ILUSRC*`/runtime/kernel/iluerror.h`', and pre-defined errors are documented in

'*ILUSRC*`/runtime/kernel/iluerrs.h`'.

Various examples of ILU protocols are available for study:

- the ONC RPC mapping ('*ILUSRC*`/runtime/kernel/sunrpc.c`'
- the XNS Courier protocol ('*ILUSRC*`/runtime/kernel/courier.c`')
- the OMG Internet Inter-Orb Protocol ('*ILUSRC*`/runtime/kernel/iiop.c`')

10.3.2 Transport Filters

In general, ILU protocols form 'messages' consisting of sequences of bytes, which are then passed to the ILU transport layer to be conveyed to another address space. The transport layer itself is composed of one or more ILU *transport filters*, each of which handles the message in turn. These filters are either communication filters, such as the filters which actually convey messages via TCP/IP or UDP/IP, or transformation filters, which alter the message and pass it to another transport filter, such as the ONC RPC record-marking filter, or the secure transport filter.

Each transport filter is either *reliable* or *unreliable*. All transformation filters are reliable; communication filters may or may not be reliable. A communication filter is reliable if it guarantees that any messages handed to it for transport will be reliably delivered to the other end of the communication connection. This in turn means that the communication mechanism used by the transport will take care of timeouts, retries, etc., internally, so that the ILU application need not worry about these itself. Unreliable communication filters are those which may require ILU participation in timeout and resending of messages to achieve reliable delivery.

Each filter is also either *boundaried* or *non-boundaried*. Boundaried filters are those which can comprehend and preserve message boundaries. Non-boundaried filters simply deal in chunks of bytes and have no way to recognize or preserve message boundaries. Various protocols and filters may have requirements as to whether the next filter below it in the communication stack is boundaried or non-boundaried.

New transport filters may be registered with the ILU kernel by calling the kernel function `ilu_RegisterTransport()`, described in '*ILU*`/runtime/kernel/iluxport.h`', with the name of a new transport filter and the address of a routine which returns an instance of the new transport object type. Implementing a new transport object type actually consists of implementing several related object types, including `ilu_TransportCreator`, `ilu_TransportClass`, and `ilu_Mooring`. These object types are defined in the file '*ILUSRC*`/runtime/kernel/iluntrnl.h`'.

As with protocols, the methods of the transport filter are considered to be 'inside' the ILU kernel, and must therefore conform to all ILU locking and error conventions. The locking conventions are discussed in '*ILUSRC*`/runtime/kernel/iluxport.h`'; the error conventions are documented in '*ILUSRC*`/runtime/kernel/iluerror.h`', and pre-defined errors are documented in '*ILUSRC*`/runtime/kernel/iluerrs.h`'.

Examples of transformation filters may be found in '*ILUSRC*`/runtime/kernel/sunrpcrm.c`', which is a boundaried filter implementing ONC RPC's TCP/IP record marking scheme, and

'*ILUSRC*`/runtime/kernel/security.c`', which is a non-boundaried filter implementing message integrity, sender authentication, and message privacy. Examples of communication filters may be found in '*ILUSRC*`/runtime/kernel/newtcp.c`', which is a non-boundaried reliable filter implementing data communication via TCP/IP, '*ILUSRC*`/runtime/kernel/udp.c`', which is a non-boundaried unreliable filter implementing data communication via UDP/IP, and '*ILUSRC*`/runtime/kernel/inmem.c`', which is a boundaried reliable filter implementing intra-address-space communication via memory buffers.

10.4 Object Incarnation Procedures

ILU true objects live in *kernel servers*, a kernel data structure that handles communication and other aspects of the object implementation. When an object reference is received from another address space, the kernel server is responsible for mapping this reference to an actual object. Normally, the kernel server simply consults an internal hash table for an object corresponding to a specified 'instance handle'; however, an application may register an application-specific callback function to be used instead. This allows on-the-fly creation of objects, which is often vital when handling many objects. Actual in-memory representations of the objects can be garbage-collected, then dynamically re-incarnated when needed by a client.

The application registers this functionality by creating an implementation of an `ilu_ObjectTable` object, and passing that implementation as a parameter to `ilu_CreateTrueServer` when creating the kernel server. Typically, `ilu_CreateTrueServer` is called directly only by a language-specific runtime; the actual application would work with object tables via whatever mechanism is exported by the language-specific runtime. Check the documentation for your particular language runtime for more information.

10.5 Malloc Failure Recovery

ILU uses a number of internal interfaces to allocate and free memory, such as `ilu_malloc()`, `ilu_free()`, and `ilu_realloc()`. These functions wrap calls to the standard `malloc()`, etc., in wrappers that allow for better error handling. They are documented in

'*ILUSRC*`/runtime/kernel/iluxport.h`'.

Applications can register callback functions to handle `malloc` failures, in two ways. The kernel function `ilu_AddFreer()` allows registration of routines which can be called to free up memory, to allow a `malloc` call to succeed. The kernel functions `ilu_SetMemFaultAction()` and `ilu_SetMemFaultConsumer()` allow applications to determine what action should be taken if a `malloc` failure occurs.

10.6 Error Reporting

ILU includes a comprehensive error-signalling system in the kernel library, which is documented in '*ILUSRC*`/runtime/kernel/iluerror.h`'. In addition, the kernel library contains many calls to `_ilu_Assert`, which check that various kernel invariants are maintained. When a runtime assertion fails, the kernel may either call an application-specified failure handler, set by a call to `ilu_SetAssertionFailureConsumer()`, or take one of three default actions, chosen by a call on `ilu_SetAssertionFailureAction()`. The three default actions are (1) to generate an illegal instruction trap, and thus coredump; (2) to exit with some error code; and (3) to enter an endless loop, calling `sleep()` repeatedly. The third action is the default action; the intent is to stop the program with all invalid data intact on the stack, and network connections intact, so that a debugger may attach to the 'live' process.

10.7 Debugging Interfaces

The ILU kernel contains a large number of debugging print statements, which document various things going on inside the kernel. The specific things printed may be controlled by calling either `ilu_SetDebugLevel()` or `ilu_SetDebugLevelViaString()`. The specific bits which can be specified to `ilu_SetDebugLevel()`, or names which can be specified to `ilu_SetDebugLevelViaString()`, are documented '*ILUSRC*`/runtime/kernel/iludebug.h`'.

All debugging messages are displayed via calls to the kernel function `ilu_DebugPrintf()`. Normally, this routine simply calls `vfprintf (stderr, ...)` to actually output the messages. However, this can be changed to call some application-specific message output system by calling `ilu_SetDebugMessageHandler()`, documented in '*ILUSRC*`/runtime/kernel/iluxport.h`'. Two special values are defined for, and accepted by, `ilu_SetDebugMessageHandler()`; the value

`ILU_DEFAULT_DEBUG_MESSAGE_HANDLER` causes the debug system to revert to the original output handler; the value `ILU_NIL_DEBUG_MESSAGE_HANDLER` causes the debug system to simply discard any debugging messages.

11 Debugging ILU Programs

This document describes some of the common errors that occur with the use of ILU, and some techniques for dealing with them.

11.1 C++ static instance initialization

Our support for C++ currently depends on having the constructors for all static instances run before `main()` is called. If your compiler or interpreter doesn't support that, you will experience odd behavior. The C++ language does not strictly mandate that this initialization will be performed, but most compilers seem to arrange things that way. We'd like to see how many compilers *do not*; if your's doesn't, please send a note to `ilu-bugs@parc.xerox.com` telling us what the compiler is.

ILU uses the static-object-with-constructor trick to effect per-compilation-unit startup code. In certain cases you'll want to ensure that a certain compilation unit's initialization is run before another's. While C++ defines no standard way to do this, most compilers work like this: compilation units are initialized (static object construtors run) in the order in which they are given to the link-editor. We (`ilu-bugs@parc.xerox.com`) want to hear about any exceptions to this rule.

11.2 ILU trace debugging

ILU contains a number of trace statements that allow you to observe the progress of certain operations within the ILU kernel. To enable these, you can set the environment variable `ILU_DEBUG` with the command `setenv ILU_DEBUG "`*xxx*`:`*yyy*`:`*zzz*`:..."` where *xxx*, *yyy*, and *zzz* are the names of various trace classes. The classes are (as of December 1995) `packet`, `connection`, `incoming`, `export`, `authentication`, `object`, `sunrpc`, `courier`, `dcerpc`, `call`, `tcp`, `udp`, `xnsspp`, `gc`, `lock`, `server`, `malloc`, `mainloop`, `iiop`, `error`, `sunrpcrm`, `inmem`, and `binding`. The special class `ALL` will enable all trace statements: `setenv ILU_DEBUG ALL`. The function `ilu_SetDebugLevelViaString(char *trace_classes)` may also be called from an application program or debugger, to enable tracing. The argument *trace_classes* should be formatted as described above.

`ILU_DEBUG` may also be set to an unsigned integer value, where each bit set in the binary version of the number corresponds to one of the above trace classes. For a list of the various bit values, see the file '*ILUHOME*`/include/iludebug.h`'. Again, you can also enable the tracing from a program or from a debugger, by calling the routine `ilu_SetDebugLevel(unsigned long trace_bits)` with an unsigned integer argument.

The routine `ilu_SetDebugMessageHandler` allows an application to specify an alternate routine to be called when an error or debugging message is to be printed.

`void` **ilu_SetDebugMessageHandler** (`void (*`*handler*`) (char *formatSpec, va_list args)`) [ILU kernel]

Locking: unconstrained

Registers *handler* with the ILU kernel to be called whenever a debugging or error message is output via `ilu_DebugPrintf`, instead of the default handler, which simply prints the message to `stderr`, using `vfprintf`. Two special constant values for *handler* are defined, `ILU_DEFAULT_DEBUG_MESSAGE_HANDLER`, which will cause the default behavior to be resumed, and `ILU_NIL_DEBUG_MESSAGE_HANDLER`, which will cause debugging and error messages to be simply, silently, discarded.

11.3 Debugging ISL

11.3.1 Use of `islscan`

The `islscan` program is supplied as part of the ILU release. It runs the ISL parser against a file containing an interface, and prints a "report" on the interface to standard output. It can therefor be used to check the syntax of an interface before running any language stubbers.

11.3.2 The ISLDEBUG environment variable

Setting the environment variable `ISLDEBUG` to any value (say, "t"), before running any ILU stubber or the program `islscan`, will cause ILU's parser to print out its state transitions as it parses the ISL file. If you're having a serious problem finding a bug in your ISL file, this might help.

11.4 Bug Reporting and Comments

Report bugs (nah! – couldn't be!) to the Internet address `ilu-bugs.parc@xerox.com`, or to the XNS address `ILU-bugs:PARC:Xerox`. Bug reports are more helpful with some information about the activity. General comments and suggestions can be sent to either `ILU@parc.xerox.com` or `ILU-bugs`.

Often the our first reply to a bug report is a request for a typescript that shows the bug occurring, with all trace debugging turned on. If that doesn't make it clear to us, our second reply may be a request for a stack trace, with printouts of relevant variables and data strutures. Including these things in your bug report may speed the cycle of interactions.

11.5 Use of `gdb`

When using ILU with C++ or C or even Common Lisp, running under the GNU debugger `gdb` can be helpful for finding segmentation violations and other system errors.

ILU provides a debugging trace feature which can be set from `gdb` with the following command:

```
(gdb) p ilu_SetDebugLevel(0xXXX)
ilu_SetDebugLevel:  setting debug mask from 0x0 to 0xXXX
$1 = void
(gdb)
```

The value *XXX* is an unsigned integer as discussed in section 3. The debugger `dbx` should also work.

We are in the midst of installing a consistent new way of handling rutime failures into the ILU runtime kernel. This new way involves the kernel reporting the failure to its caller; the old way involves combinations of panicking, reporting to the user (not the caller) via a printed message, and fragmentary reporting to the caller. Every time a runtime failure is noted the new way, the procedure `_ilu_NoteRaise` in '*ILUSRC*`/runtime/kernel/error.c`' is called; this procedure thus makes a good place to set a breakpoint when debugging. Most runtime failures occur due to genuine problems; some occur during normal processing (e.g., end-of-file detection).

11.6 Error handling

Ideally, the ILU runtime would report all failures to the application, in the way most appropriate for the application's programming language. Sadly, this is not yet the case.

The ILU runtime kernel has three kinds of runtime failures:

1. memory allocation failures from which the kernel cannot proceed;
2. internal consistency check failures, from which the kernel cannot proceed; and
3. internal consistency check failures, which the kernel is prepared to report to the ILU language-specific runtime veneer (which, hopefully, would in turn report the failure to the applicaiton).

The second kind is being eliminated. The first kind is being reduced, and might also be eliminated.

The application can specify how each of these three kinds of runtime failures is to be handled. The choices are:

1. Print an explanatory message and then explicitly trigger a `SEGV` signal by attempting to write to protected memory. This is useful for generating core dumps for later study of the error.
2. Print an explanatory message and then exit the program with an application-specified exit code.
3. Print an explanatory message and then enter an endless loop, which calls `sleep(3)` repeatedly. This option is useful for keeping the process alive but dormant, so that a debugger can attach to it and examine its "live" state. This is the default action for all three kinds of failures.
4. Invoke an application-supplied procedure (without printing anything first).

5. Report the failure out of the kernel, without printing anything first (this option is available only for the third kind of failure).

An application can change the action taken on memory failures by calling `ilu_SetMemFailureAction` or `ilu_SetMemFailureConsumer`.

`void` **ilu_SetMemFailureAction** (`int` *mfa*) [ILU kernel]

Locking: unconstrained

Calling this tells the ILU kernel which drastic action is to be performed when `ilu_must_malloc` fails. -2 means to print an explanatory message on stderr and then coredump; -1 means to print an explanatory message on stderr and then loop forever in repeated calls to `sleep(3)`; positive numbers mean to print an explanatory message on stderr and then `exit(`*mfa*`)`. The default is -1.

`typedef void (*) (const char *file, int line)` **ilu_FailureConsumer** [ILU kernel]

A procedure that is called when the ILU kernel can't proceed. This procedure must not return.

`void` **ilu_SetMemFailureConsumer** (`ilu_FailureConsumer` *mfc*) [ILU kernel]

Locking: unconstrained

An alternative to `ilu_SetMemFailureAction`: this causes *mfc* to be called when `ilu_must_malloc fails`.

Similarly, an application specifies how unrecoverable runtime consistency check failures are to be handled by calling `ilu_SetAssertionFailureAction` or `ilu_SetAssertionFailConsumer`, which are exactly analogous to the procedures for memory failure handling. For recoverable consistency check failures, an application can call `ilu_SetCheckFailureAction` or `ilu_SetCheckFailureConsumer`.

`void` **ilu_SetCheckFailureAction** (`int` *cfa*) [ILU kernel]

Locking: unconstrained

Calling this tells the runtime which action is to be performed when an internal consistency check fails. -3 means to raise an error from the kernel (without necessarily printing anything); -2 means to print an explanatory message to stderr and then coredump; -1 means to print and then loop forever; non-negative numbers mean to print and then `exit(`*cfa*`)`; others number reserved. The default is -1.

`typedef void (*) (const char *file, int line)` **ilu_CheckFailureConsumer** [ILU kernel]

A procedure for handling an internal consistency check failure. If this procedure returns, the consistency check failure will be raised as an error from the kernel.

`void` **ilu_SetCheckFailureConsumer** (`ilu_CheckFailureConsumer` *cfc*) [ILU kernel]

Locking: unconstrained

An alternative to `ilu_SetCheckFailureAction`: this causes *cfc* to be called (and no printing); if *cfc* returns, an error will be raised from the kernel.

11.6.1 Decoding reportable consistency check failures

For language mappings consistent with CORBA, the third kind of failure is reported as an occurrence of the CORBA system exception `internal`, with a minor code that encodes the filename and line number where the consistency check occurs. The coding is this: 10,000*hash(filename, 32771) + linenum + 1,000. The directory part, if any, is stripped from the filename before hashing. To aid in decoding these minor codes, ILU includes the program `decoderr`, which is used like this:

```
% decoderr 269211234
269211234 = line 234, file $ILUSRC/runtime/kernel/call.c
```

If a reportable consistency check failure occurs in a file not anticipated in the construction of `decoderr`, you'll see something like this:

```
% decoderr 60612345
60612345 = line 1345 in unknown file (that hashes to 6061)
```

The program `iluhashm` can be used to hash given filenames, so you can search a set of candidates for the mysterious hash code:

```
% iluhashm 32771 ../cpp/foobar.cpp ../cpp/barfoo.cpp
/* Generated at Mon Dec 11 22:44:47 1995
   with modulus 32771 */
{      6061, "../cpp/foobar.cpp"},
{     13273, "../cpp/barfoo.cpp"},
```

12 Installation of ILU

This document describes the installation of version RELEASE_NUMBER of the Inter-Language Unification (ILU) system.

If you succeed in installing ILU on a particular platform, we'd appreciate it if you could send a note to `ilu-core@parc.xerox.com` *telling us (1) what operating system you succeeded with, and what version of that OS, (2) which versions of what compilers you used, and (3) which version of ILU you used. We're accumulating a list of operating systems and compilers that work with ILU. If you had to make any changes to make it work on your system, please send them along, and we'll incorporate them into the next release.*

12.1 Installing on a UNIX System

12.1.1 Prerequisites

You will need an ANSI C compiler to build and install ILU, along with an ANSI C-compliant '`libc.a`'. Note that GNU `gcc` doesn't always work as an ANSI C compiler. The simple test we use to qualify a compiler is whether it can compile and link the following program without warnings or errors:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
int main(int ac, char **av) {
 int i = INT_MAX;
 char *p = (char *) malloc(1048);
 memmove(p, *av, strlen(*av)+1);
 printf("%s %d\n", p, i);
 return 0;
}
```

ILU requires the `imake` program from the MIT X Consortium release of the X Window System, version 4 or later. This is available via FTP from the ftp servers `ftp.x.org` on the East Coast, or `gatekeeper.dec.com` on the West Coast. You can also get it from

`ftp://ftp.parc.xerox.com/pub/ilu/imake/imake.tar.gz`.

ILU normally provides support for a number of languages, currently ANSI C, C++, Modula-3, Python, and Common Lisp (Franz Allegro 4.2, and Harlequin LispWorks), but the 2.0alpha release only contains support for ANSI C, C++, and Python.

- If you wish to build the support for ANSI C, you will need a C compiler, and an ANSI C-compliant libc. But you already need that to build ILU.

- If you wish to build the support for C++, you will need a C++ compiler that conforms to version 2.0 of the C++ specification. ILU does not use either C++ templates or exceptions, as these are too spottily implemented to be relied on. The GNU C/C++ compiler `g++` seems to work well with ILU. It has also been tested with Lucid's Energize `lcc` compiler, CenterLine's `CC` compiler, and Sun's `CC` compiler.
- If you wish to build support for Modula-3, you will need to wait for the 2.0beta release.
- If you wish to build the support for Common Lisp, you will need to wait for the 2.0beta release.
- If you with to build support for Python, you will need the Python 1.1.1 (or later) release, available via FTP from '`ftp://ftp.cwi.nl/pub/python/`'.

12.1.2 For the Impatient

You can try just unpacking it, and typing

```
% make
```

at the top of the source tree. A minimal configuration using defaults will be built, sufficient for testing.

12.1.3 Real Configuration

Begin by creating two directories: one, *ILUHOME*, to install the ILU in, and the other, *ILUSRC*, to unpack the sources in, and build the system in. It is often convenient if *ILUSRC* is a subdirectory of *ILUHOME*, but it is not necessary. At PARC, we use '`/import/ilu`' for *ILUHOME*, and '`/import/ilu/src`' for *ILUSRC*.

Copy the compressed tar file '`ilu-RELEASE_NUMBER.tar.Z`' or '`ilu-RELEASE_NUMBER.tar.gz`' to *ILUSRC*. Uncompress it if necessary with the `uncompress` or `gunzip` program:

```
% uncompress ilu-RELEASE_NUMBER.tar.Z
```

or

```
% gunzip ilu-RELEASE_NUMBER.tar.gz
```

Then unpack the tar file:

```
% tar xf ilu-RELEASE_NUMBER.tar
```

ILU uses the GNU autoconf system to configure the release, before building. The very simplest way to configure your system is type type

```
% ./configure
```

at the top of the source tree.

What will happen is that *ILUSRC*`/imake/configure` will go out and look along the value of your PATH environment variable for various executable programs. If it finds `cc`, it will assume that you want to build ANSI C support for ILU. If it finds `m3`, it will assume that you want to build Modula-3 support for ILU. If it finds `cl` or `franz`, it will assume that you want to build Lisp support for ILU. If it finds `python`, it will assume that you want to build Python support for ILU. If it finds `CC` or `cxx` or `c++`, it will assume that you want to build C++ support for ILU. It will also assume that the first executable with an appropriate name is the one you wish to use for compiling programs in that language. By default, it will assume that you wish to include support only for using `ONC RPC` over `TCP/IP`. By default, it will assume that you do not wish to provide support for OMG IDL.

`imake` must be on your path, or in '`/usr/bin/X11/`', for the build to work properly.

12.1.3.1 `configure` Options

The program `configure` can be invoked with a number of options, to customize the build for your site:

- `--with-destdir=PATH` – this should be the name of the directory you would like the files installed into, in the ***make Install*** step. The default is '`/usr/local/ilu`', which may be OK for your system.
- `--with-iluhome=PATH` – this should be the name of the directory users will expect ILU to be installed under, the value of *ILUHOME*. The two options **`--with-destdir`** and **`--with-iluhome`** are provided separately because a directory may have two different names that are used to access it in different ways. At PARC, for instance, installation directories are often write-protected if named with their ordinary names, and a special name has to be given to enable writing in that directory. If your site does not have this type of restriction, the switches for **`--with-destdir`** and **`--with-iluhome`** should probably have the same values. Note that the default is '`/usr/local/ilu`', which may be OK for your system.
- `--prefix=PATH` – if specified, this becomes the default for *ILUHOME* and *DESTDIR*, if they are not specified explicitly.
- `--with-binding-dir=PATH` – a world-writable directory in a file system, for supporting the "simple binding" service described in Chapter 1.
- `--enable-sunrpc-protocol` – compiles in support for using `ONC RPC` across address spaces. (Enabled by default.)
- `--enable-courier-protocol` – compiles in support for using `XNS Courier RPC` across address spaces. (Disabled by default)
- `--enable-tcp-transport` – compiles in support for using `TCP/IP` to transport messages. (Enabled by default.)
- `--enable-udp-transport` – compiles in support for using `UDP/IP` to transport messages. (Disabled by default)

- `--enable-omg-idl-support` – builds the support for using OMG IDL with ILU. You will need to have a C++ compiler (CFRONT 3.0 or later) on your path. `g++` seems to work fine. (Disabled by default)
- `--enable-xview-support` – builds the libraries to interface the XView GUI library to the ILU main loop. (Disabled by default)
- `--enable-xt-support` – builds the libraries to interface Xt-based GUI libraries to the ILU main loop. (Disabled by default)
- `--enable-tk-support` – builds the libraries to interface Tk-based GUI libraries to the ILU main loop. (Disabled by default)
- `--enable-c-support` – enables C support. Use `--disable-c-support` to turn off the default support for C.
- `--enable-cplusplus-support{=`*PROG*`}` – enables C++ support. If *PROG* is specified, treats *PROG* as the command to use for C++ compilation. Default is to enable C++ support iff the environment variable *CXX* is defined, in which case *CXX* should be the compilation command.
- `--enable-modula3-support{=`*M3-COMMAND*`}` – enables Modula-3 support. If *M3-COMMAND* is specified, treats *M3-COMMAND* as the command to use for Modula-3 compilation, otherwise looks for the program `m3` on your path, and use the full path name for that program. Default is disabled.
- `--enable-lisp-support{=`*PROG*`}` – enables Common Lisp support. If *PROG* is specified, treats *PROG* as the command to use to invoke Common Lisp. Default is disabled.
- `--enable-python-support{=`*PYTHON-HOME*`}` – enables Python support. If *PYTHON-HOME* is specified, treats *PYTHON-HOME* as the directory in which the Python system is installed. Default is enabled.

Features that are enabled by `--enable-FEATURE` may also be explicitly disabled with `--disable-FEATURE`. The particular ANSI C compiler to use may be specified by setting the environment variable `CC` before running `configure`.

12.1.4 Building

Now that you have configured the release, do the following to build the system. Note that the capitalization of the arguments to `make` is important.

1. Set your working directory to *ILUSRC*:

   `% cd` *ILUSRC*

2. Build the system with the command:

   `% make`

3. If the build goes well, install the system with the command

   `% make Install`

4. After the installation is complete, you may remove extra files in *ILUSRC* with the command

   ```
   % make Clean
   ```

   You may wish to use `make Clean` at any time, to get your system into a consistent state.
5. If you change the configuration files, you should clean the system with the command '`make Clean`', and redo the installation starting at step 2. If you run into problems that can be fixed without changing the configuration files, you can re-build the system by starting at step 3.

12.1.5 Environment Variables

ILU tools use a number of UNIX environment variables under the covers.

- The variable `ILUHOME` should be set to point to the value of *ILUHOME*.
- The variable `ILUPATH` should be set to a colon-separated list of directories in which the tools look for interface files. A minimal value for `ILUPATH` is probably `.:${ILUHOME}/interfaces`.
- Your `PATH` environment variable should have the directory '*ILUHOME*`/bin`' on it.
- Your `MANPATH` variable should have the directory '*ILUHOME*`/man`' on it.
- If you are using Common Lisp, the portable DEFSYSTEM included with ILU uses the value of `SYSDCLPATH` to find system descriptions. It should be a colon-separated list of directories. A good initial value might be `.:${ILUHOME}/lisp`. See Appendix A of the reference manual for more details on the portable DEFSYSTEM.
- If you are using Python, the value of the environment variable `PYTHONPATH` should include the directory in which the `ilu` library for Python has been installed; that's normally '*ILUHOME*`/lib`'.
- The variable `ISLDEBUG` can optionally be set to any value to enable tracing in the ISL parser.
- The variable `ILU_DEBUG` can be optionally be set to a colon-separated list of trace values to enable tracing in the ILU runtime kernel. See Section 11.6.1 [Debugging ILU Programs], page 109, for more information.

12.1.6 Testing the Build

There are several ways to test the build. The most straightforward is to build and install it somewhere. Set up your environment variables as described above. Then make a scratch directory, which we'll refer to as *TESTDIR*, and do the following:

```
% cd TESTDIR
% cp ILUHOME/examples/test1/* .
% ilumkmf
% make client server
ILUHOME/bin/c-stubber  Test1.isl
header file for interface Test1 to ./Test1.h...
common code for interface Test1 to ./Test1-common.c...
code for surrogate stubs of interface Test1 to ./Test1-surrogate.c...
```

```
code for true stubs of interface Test1 to ./Test1-true.c...
ILUHOME/bin/c-stubber  Test2.isl
header file for interface Test2 to ./Test2.h...
common code for interface Test2 to ./Test2-common.c...
code for surrogate stubs of interface Test2 to ./Test2-surrogate.c...
code for true stubs of interface Test2 to ./Test2-true.c...
ILUHOME/bin/c-stubber  Test3.isl
header file for interface Test3 to ./Test3.h...
common code for interface Test3 to ./Test3-common.c...
code for surrogate stubs of interface Test3 to ./Test3-surrogate.c...
code for true stubs of interface Test3 to ./Test3-true.c...
rm -f clnt.o
cc -c -g -I. -IILUHOME/include  clnt.c
rm -f Test1-surrogate.o
cc -c -g -I. -IILUHOME/include  Test1-surrogate.c
rm -f Test1-common.o
cc -c -g -I. -IILUHOME/include  Test1-common.c
rm -f Test2-surrogate.o
cc -c -g -I. -IILUHOME/include  Test2-surrogate.c
rm -f Test2-common.o
cc -c -g -I. -IILUHOME/include  Test2-common.c
rm -f Test3-surrogate.o
cc -c -g -I. -IILUHOME/include  Test3-surrogate.c
rm -f Test3-common.o
cc -c -g -I. -IILUHOME/include  Test3-common.c
rm -f client
cc -g -o client clnt.o Test1-surrogate.o Test1-common.o \
  Test2-surrogate.o Test2-common.o Test3-surrogate.o \
  Test3-common.o  ILUHOME/lib/libilu-c.a \
  ILUHOME/lib/libilu.a
rm -f srvr.o
cc -c -g -I. -IILUHOME/include  srvr.c
rm -f Test1-true.o
cc -c -g -I. -IILUHOME/include  Test1-true.c
rm -f Test3-true.o
cc -c -g -I. -IILUHOME/include  Test3-true.c
rm -f server
cc -g -o server srvr.o Test1-common.o Test1-true.o \
  Test2-common.o Test3-common.o Test3-true.o \
  ILUHOME/lib/libilu-c.a ILUHOME/lib/libilu.a
% ./server &
[1] 7079
% exported ilu:Test1-Server/Test1_Initial_Object;ilu%3AiX2w6hjR-...
% ./client
Test1.01.U-CSS-to-U
u._d=5, u._u.boolean = 1, u._u.01 = 0x1ffee7c
Test1.01.f-CSS-to-R0
ro->i=9
Test1.01.R-ScS-to-F
f=39.700001
```

```
Test1.01.a-RO
Test1.01.get-02
got 02, sbh = ilu:Test1-SunRPC-Server/1;ilu%3AaUtts57Ywbp2fxe6+-...
Test1.o2.00-A0-to-CSS
Test1.02.R-I-A1-to-I-A0
Test1.01.get-03
making 03...
got 03, sbh = ilu:Test1-Server/2;ilu%3An+eRrvAZ8JB9v2qoX7sJGPxdX...
Test1.03.RS-R-to-R-IS
Test1.03.01-U-to-U
u._d=3, u._u.boolean = 0, u._u.01 = 0xd2b78
Test1.01.get-03
got 03, sbh = ilu:Test1-Server/3;ilu%3Ab-mNa9uj0TsJAp7YrlEh0AUfX...
Test3.0.RS-R-to-R-IS
Test3.0.01-U-to-U(0xd7520, {3})
u._d=3, u._u.boolean = 0, u._u.01 = 0xd2b78
Test3.0.I-to-Test1U(397)
Test3_0_I_to_Test1U:  u2._d=5, u2._u.boolean = 1, u2._u.01 = 0x10a88d0
Test1.01.get-03
making 04...
got 03, sbh = ilu:Test1-Server/4;ilu%3Ad8sZGQLLpVsJ2PBL5BoIX45q0...
Test1.04.R_to_R (12345.6789000000) => 1020304.0506070800
doubles:  r1 is 12345.6789000000, r2 is 1020304.0506070800
%
```

You can proceed to test the various other clients and servers in different languages against each other. See the file '*ILUHOME*/`examples/test1/README`' for more information.

12.1.7 Notes on Specific Systems

12.1.7.1 DEC ALPHA with OSF OS:

From `hassan@db.stanford.edu`: "Use `cc` instead of `gcc` as the C compiler, and make sure to include the '-taso' switch."

From `jg@w3.org`: "I built [ILU 2.0 alpha on OSF 3.2B] without the '-taso' switch. Is this still needed? `c-stubber` certainly ran without it this release."

12.1.7.2 SunOS 4.1.x

Note that the default Sun C compiler is not ANSI C, nor is `gcc` when installed against the normal Sun header files and '`/lib/libc.a`'. You will have to use either `gcc` with the GNU C Library glibc, or the SunPro ANSI C compiler `acc`, or Lucid Energize `lcc`, or some other ANSI compiler.

12.2 Examples

The following example uses of ILU are provided in the installed tree as subdirectories of '*ILUHOME*/`examples/`':

- '`tutorial`' – this contains the code for a small ILU tutorial, implemented in both ANSI C and Python. If you are interested in using either of these languages with ILU, we suggest starting with the code in this directory, and one of the files '*ILUHOME*/`doc/tutorial-C.ps`' and '*ILUHOME*/`doc/tutorial-Python.ps`'. Even if you want to use a different programming language, it's probably worth your while reading one of these tutorials.
- '`test1`' – this contains our basic interoperability test program, implemented in a number of languages. If you are interested in working with Common Lisp, Modula-3, or C`++`, we recommend starting here.
- '`multlang`' – an example of using multiple languages (ANSI C and Python) in the same address space, communicating via ILU.
- '`blob`' – a contributed Python example.
- '`fs`' – an object-oriented file server, written in Common Lisp.
- '`foogen`' – an example of using ILU with Modula-3.
- '`timeit`' – a crude timing test.
- '`test2`' – ANSI C programs to test ASYNCHRONOUS methods and concurrent protocols.
- '`changeup`' – a test case to simulate client-server situations where connections are frequently being broken and restored.
- '`dialog`' – a simple test program for simulated network failures.

Read the '`README`' file in each directory first.

12.3 Name Servers

No standard "name service" or binding service is provided. We feel that this is an area to be addressed independently, and we may include a name service in future releases of ILU. An experimental simple name service bootstrap interface is available as the *simple binding system*. See the ANSI C `ILU_C_PublishObject`, `ILU_C_WithdrawObject`, and `ILU_C_LookupObject`, and corresponding routines in the other languages, for more details. This interface is not guaranteed to be present in future releases.

12.4 Documentation

ILU documentation is provided in a pre-formatted form, PostScript. The source form of the documentation is called TIM, and is documented in the ILU reference manual. If for some reason you do need to rebuild the documentation, you should have the systems TEX, Perl, ghostscript, `dvips`, and pbmplus; if you can't find these yourself, please send mail to `ilu-core@parc.xerox.com` for info on how to find them.

12.5 Bug Reporting and Comments

12.5.1 Known Bugs and Gotchas

KNOWN BUGS:

Release 2.0alpha:

1) [C++] The RemoveAll method on sequence objects uses a quadratic algorithm rather than possible linear algorithm.

2) [configuration] Watch out for use of predefined Imake (i.e., cpp) macros in local.defs! Common boobytraps include names of processors, vendors, and operating systems (e.g., sparc, sun), which are used (as isolated tokens according to C rules) in some folks' conventions for naming directories. If you're lucky, you can solve these problems with quoting. A more heavy-duty approach is to #undef the offending macros at the start of local.defs, and re#define them at the end.

3) [kernel] There's no good way to control UDP timeouts. The bad way isn't declared in any header file.

4) [ISL] "TYPE X = OBJECT SUPERTYPES Y ...; TYPE Y = OBJECT SUPERTYPES X ..." crashes the parser (and thus islscan and all the stubbers). Don't do this!

5) [kernel] FoundFD (and TAInvoke) in mainloop.c should unregister I/O hander. There's a patch for this against ILU 1.8 in the 1.8 bugs file.

6) [kernel] Default_Unregister{Input,Output}Source should clear bits. There's a patch for this against ILU 1.8 in the 1.8 bugs file.

7) [MOP] If ilu_AddRegisterersToDefault is used, callers of ilu_Register{Input,Output}Source must be prepared for false callbacks.

8) [C++] iluServer::iluSetMainLoop doesn't pass an ilu_MainLoop with NULLs in the right places in the threaded case.

9) [C++] stubber bugs noted in 1.8 bugs file have not been addressed.

12.5.2 Reporting Bugs

Report bugs (nah! – couldn't be!) to the Internet address `ilu-bugs.parc@xerox.com`, or to the XNS address `ILU-bugs:PARC:Xerox`. Bug reports are more helpful with some information about the activity; *please* read Section 11.6.1 [Debugging ILU Programs], page 109, for more information on how to look at problems. General comments and suggestions can be sent to either `ILU@parc.xerox.com` or `ILU-bugs`.

13 Multiple Languages In One Address Space

With ILU version 2.0 or later, modules implemented in different programming languages can be mixed in the same address space, with ILU doing automatic translation between data representations.[1] There are a number of things to consider when doing this; this section discusses some of them.

13.1 Dueling Runtimes

Some languages simply cannot be mixed in the same address space because their runtimes will conflict. ILU offers no solutions to this problem. Typical examples are two languages like Franz Allegro Common Lisp and DEC SRC Modula-3. They each implement a user-level threads package, and their implementations of threads probably cannot co-exist.

A possible solution to this problem, called the POSIX Portable Common Runtime (PPCR), is available from Xerox PARC, as `ftp://ftp.parc.xerox.com/pub/ppcr/`. It contains a basic runtime which can be used as the platform for a particular language implementation's runtime. Languages which use PPCR will have a lower chance of having conflicting runtimes.

13.2 Module Initialization

Module initialization really consists of two operations: interface initialization and object instantiation. The first operation initializes all the interfaces used or exported by the module; the second creates one or more true instances of objects to be used by other modules.

Generally, each ILU interface must be initialized. The process of doing this initialization varies from programming language to programming language. In ANSI C, ILU requires explicit calls to *interface*`__Initialize()` for interfaces being used, or *interface*`__InitializeServer()` for interfaces being exported. In C++, interface initialization is performed automatically, but at some indeterminate time before the first symbol from that interface is referenced from outside the interface. In Python or Common Lisp, interface initialization is performed automatically by the language at the time the module describing the interface is "loaded" into the address space.

In addition to initializing all interfaces being used or exported, a module must create one or more true object instances, to allow other modules to access it. Again, the specific way of doing this varies from programming language to programming language. Once the true instance has been created, it can be *exported* from the module by either publishing it, via the ILU simple binding

[1] Currently, it does this by marshalling one language's representation into a data buffer, as if to transmit it to another address space, then hands this buffer to the other language's ILU support, which unmarshalls it into representations appropriate for that language. We plan to optimize this process in the future.

system, or taking its string binding handle, and passing that outside the module for other modules to use.

When multiple languages are used in the same address space, each must be initialized according to the standards used for that programming language. This can be tricky when using both statically compiled and dynamically compiled languages together. Consider the case where Python and ANSI C are linked together. This use of Python may be as an extension language to a program written in C. In this case, the C code must do all initialization of modules written in C before calling into any Python module which might reference them. Similarly, Python initialization (`import`) of modules must occur before the C code can use them. See '*ILUSRC*`/src/examples/multlang/`' for an example of a situation of this sort.

In the other case, C true modules which are to be used from a Python program in the same address space must somehow be first loaded into that address space, then initialized. The loading is done by turning the C module into a Python *extension module*, and either linking it into the `python` image, or creating a dynamically loadable module from it. The initialization is done by then calling `import` on that module from within the Python interpreter. The extension module's initialization routine initializes all of its interfaces, creates one or more true objects, and exports them. After the `import` statement returns, the objects are available for finding (see next section) from within Python.

13.3 Finding Objects

Object instances may be located by calls on the variations of `LookupObject` and `ObjectOfSBH` that exist in the various language runtimes. `LookupObject` is implemented so that it first looks to see if the true object for the specified object ID is available in the local address space. If so, it returns a version of that object. Only if the object is not locally available does it perform external lookups in an attempt to locate the object. Note that for an object to be found via `LookupObject`, the true instance must first have been published via the implementation language ILU runtime's variant of `PublishObject`. If you do not want your objects published outside your address space, you should use `ObjectOfSBH` to find them.

14 Using Imake with ILU

ILU uses the `imake` system from the X Window System distribution. `imake` provides a parameterized way of constructing '`Makefile`'s automatically from '`Imakefile`'s. The '`Imakefile`'s contain macros which are expanded to regular '`Makefile`' rules when the program `imake` is run.

14.1 Creating '`Makefile`'s from '`Imakefile`'s

The program `ilumkmf` is supplied with the ILU system. When run, it will use the '`Imakefile`' in your current working directory as input, and produce the corresponding '`Makefile`', again in the current working directory:

```
% cd myilu
% ls
Imakefile foo.isl fooProg.cc
% ilumkmf
% ls
Imakefile Makefile foo.isl fooProg.cc
%
```

14.2 ANSI C Usage

A typical '`Imakefile`' for an ANSI C ILU application would look like:

```
NormalObjectRule()  /* this rule defines the .c -> .o step */

InterfaceTarget(foo.isl)
ILUCTarget(foo.h foo-surrogate.c foo-common.c foo-true.c, foo.isl)

DepObjectTarget(programComponent1.o, foo.h somethingElse.h)
ObjectTarget(programComponent2.o)

CProgramTarget(program, programComponent1.o programComponent2.o foo-surrogate.o
foo-common.o,,)
```

14.2.1 ANSI C ILU `imake` Macros

The variable `LOCAL_INCLUDES` is a list of include file locations to be included when compiling.

The variable `ANSI_C_COMMAND` defines the particular command invoked for compiling ANSI C on your system. If you wish to use a different ANSI C compiler, override the default command by redefining this value in your '`Imakefile`'. Note that it may also be necessary to build a version of the ILU ANSI C library, '*ILUHOME*`/lib/libilu-c.a`', to use with this compiler.

`NormalObjectRule()` defines a number of suffix rules, in particular the one to go from '`.c`' files to '`.o`' files in your environment.

`InterfaceTarget(`*ISL-file*`)` defines a number of rules about the '`.isl`' file *ISL-file*. You should have one of these in your '`Imakefile`' for every interface you use.

`ILUCTarget(`*generated-files*`,` *ISL-file*`)` defines which ANSI C files are generated from the '`isl`' file and may therefore by re-generated at will, and when the '`.isl`' file changes. Generally, for an interface called `foo`, the generated files will be '`foo-surrogate.c`', '`foo-true.c`', '`foo-common.c`', and '`foo.h`'.

`ObjectTarget(`*object-file*`)` simply states that the specified *object-file* should be built.

`DepObjectTarget(`*object-file*`,` *dependencies*`)` says that the specified *object-file* should be built, and that it depends on the files specified in *dependencies*, which is a list of file names separated by spaces. Whenever something in the *dependencies* list changes, the *object-file* will be re-built.

`CProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, so that it will be re-built if anything changes in those two groups. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. Note that the ILU ANSI C libraries are not automatically included by this command, but may be specified as part of the program by specifying them as part of either `dep-libraries` or `non-dep-libraries`.

`ILUCProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, and the normal ILU ANSI C libraries, so that it will be re-built if anything changes in those three groups, all of which will be linked into the program *program-name*. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. This differs from `CProgramTarget` in that the ILU libraries are automatically included.

14.3 C++ Usage

A typical '`Imakefile`' for a C`++` application and ILU would look like:

```
LOCALINCLUDES = -I$(ILUHOME)/include
ILULIBS = $(ILUHOME)/lib/libilu-c++.a $(ILUHOME)/lib/libilu.a

NormalObjectRule()  /* this rule defines the .cc -> .o step */

InterfaceTarget(foo.isl)
ILUCPlusPlusTarget(foo.H foo.cc foo-server-stubs.cc, foo.isl)

DepObjectTarget(programComponent1.o, foo.H somethingElse.H)
ObjectTarget(programComponent2.o)

CPlusPlusProgramTarget(program, programComponent1.o programComponent2.o foo.o,
$(ILULIBS),)
```

14.3.1 C++ ILU imake Macros

The variable `LOCAL_INCLUDES` is a list of include file locations to be included when compiling. `-I$(ILUHOME)/include` should always be on this list for compiling ILU applications.

The variable `CPLUSPLUS_COMMAND` defines the particular command invoked for compiling C++ on your system. If you wish to use a different C++, override the default command by re-defining this value. Note that it will also be necessary to build a version of ILU C++ library, '*ILUHOME*`/lib/libilu-c++.a`', to use with this compiler.

`NormalObjectRule()` defines a number of suffix rules, in particular the one to go from '`.cc`' files to '`.o`' files in your environment.

`InterfaceTarget(`*ISL-file*`)` defines a number of rules about the '`.isl`' file *ISL-file*. You should have one of these in your '`Imakefile`' for every interface you use.

`ILUCPlusPlusTarget(`*generated-files*`,` *ISL-file*`)` defines which C++ files are generated from the '`isl`' file and may therefore by re-generated at will, and when the '`.isl`' file changes. Generally, for an interface called `foo`, the generated files will be '`foo.cc`', '`foo.H`', and '`foo-server-stubs.cc`'.

`ObjectTarget(`*object-file*`)` simply states that the specified *object-file* should be built.

`DepObjectTarget(`*object-file*`,` *dependencies*`)` says that the specified *object-file* should be built, and that it depends on the files specified in *dependencies*, which is a list of file names separated by spaces. Whenever something in the *dependencies* list changes, the *object-file* will be re-built.

`CPlusPlusProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, so that it will be re-built if anything changes in those two groups. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. Note that the ILU ANSI C libraries are not automatically included by this command, but may be specified as part of the program by specifying them as part of either `dep-libraries` or `non-dep-libraries`.

`ILUCPlusPlusProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, and the normal ILU ANSI C libraries, so that it will be re-built if anything changes in those three groups, all of which will be linked into the program *program-name*. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. This differs from `CProgramTarget` in that the ILU libraries are automatically included.

14.4 Modula-3 Usage

A typical '`Imakefile`' for a Modula-3 application and ILU would look like:

```
LOCALM3FLAGS = -D$(ILUHOME)/include -L$(ILUHOME)/lib
```

```
InterfaceTarget(foo.isl)
ILUM3Target(IluM3Files(foo), foo.isl)
M3LibraryTarget(libfoo.a, IluM3Files(foo), -lilu-m3)
M3ProgramTarget(FooM3Server, FooM3Server.m3 libfoo.a, -lilu-m3 -lilu)
M3ProgramTarget(FooM3Client, FooM3Client.m3 libfoo.a, -lilu-m3 -lilu)
```

14.4.1 Modula-3 ILU `imake` Macros

The variable `LOCALM3FLAGS` is a list of extra arguments to be passed to the `m3` command when compiling. `-D$(ILUHOME)/include` and `-L$(ILUHOME)/lib` should always be on this list for compiling ILU applications.

The variable `M3_COMMAND` defines the particular command invoked for compiling Modula-3 on your system. If you wish to use a different Modula-3, override the default command by redefining this value. Note that it will also be necessary to build a version of ILU Modula-3 library, '*ILUHOME*`/lib/libilu-m3.a`', to use with this compiler.

`IluM3Files(`*base*`)` expands to the series of filenames generated by the Modula-3 stubber from '*base*`.isl`': '*base*`.i3` *base*`_x.i3` *base*`_y.m3` *base*`_c.m3` *base*`_s.m3`'.

`ILUM3Target(`*generated-files*`,` *ISL-file*`)` declares that files *generated-files* are generated from the '`isl`' file and may therefore by re-generated at will, and should be when the '`.isl`' file changes.

`M3LibraryTarget(`*library-name*`,` *parts*`,` *non-dep-libraries*`)` defines a library called *library-name* that is built from *parts* and *non-dep-libraries*; it will be re-built if anything among the *parts* changes. The *library-name* should be the name of the '`.a`' file, including the ".a".

`M3ProgramTarget(`*program-name*`,` *parts*`,` *non-dep-libraries*`)` defines a program called *program-name* that is built from *parts* and *non-dep-libraries*; it will be re-built if anything among the *parts* changes.

15 The TIM Documentation Language

This document describes the TIM documentation language that the documentation for ILU is written in. It is not necessary to be familiar with TIM to use ILU; you will only need to know TIM if you wish to use it to write or modify documentation.

15.1 TIM

TIM is essentially a superset of the GNU texinfo language, version 2. It adds several features to allow more precise discrimination of semantics when documenting software systems. You should be familiar with the basic texinfo system first. Documentation on texinfo is supplied with the ILU distribution; you should be able to find it in the files '`ilu/doc/texinfo2.ps`'.

TIM removes the need to begin every file with `\input texinfo`, and to end every file with `@bye`. These lines are added automatically by TIM as needed. This allows a file to define both a stand-alone document, and to be included as a section in some larger document.

TIM redefines the following texinfo markup commands:

- `@var` is now used to indicate a regular programming language variable. The term `@metavar` is used to mark meta-variables.

TIM also extends texinfo by adding the following markup:

- `@C` is used to mark artifacts of the C language, e.g., `@C{#define}`.
- `@C++` is used to mark artifacts of the C++ language, e.g., `@C++{#define}`.
- `@class` is used to mark names of object classes.
- `@command` is used to mark user input, such as a user-typed shell command, when it occurs in the normal flow of text. The term `@userinput` is used when the user input occurs within a `@transcript` section.
- `@codeexample` is used to mark code that is excerpted in the style of a texinfo `example`. The term `@codeexample` should appear on a line by itself, before the text of the code, and the terms `@end codeexample` should appear on a line by itself, at the end of the text of the code.
- `@cl` is used to mark artifacts of the Common Lisp language, e.g., `@cl{defmacro}`.
- `@constant` can be used to mark constant names and values that appear in the text.
- `@exception` is used to mark names of exceptions.
- `@fn` is used to mark function names that occur in the text.
- `@interface` is used to mark interface names.
- `@isl` is used to mark artifacts of the ILU ISL language, e.g. `@isl{SIBLING}`.
- `@kwd` is used to mark keywords that occur in the text.
- `@language` is used to mark names of computer or human languages.

- `@m3` is used to mark artifacts of the Modula-3 language, e.g. `@m3{INTERFACE Foo;}`.
- `@macro` is used to mark names of macros that occur in the text.
- `@message` is used to mark in-line text that is a message a program may write to its output.
- `@metavar` is used to mark meta-variables.
- `@method` is used to mark method names.
- `@module` is used to denote module names for those languages which support them, such as Common Lisp package names, or Modula-3 module names.
- `@parm` is used to mark parameter names.
- `@picture` is used to include a file containing encapsulated Postscript of a diagram or picture. It should appear on a line, followed by the name of the file containing the picture, followed by a newline. We find the InterViews tool `idraw` works well in creating diagrams in the form of encapsulated Postscript.
- `@program` is used to mark program names that occur in the text.
- `@protocol` is used to mark names of ILU RPC protocols.
- `@system` is used to mark system names that occur in the text.
- `@transcript` is used to mark an example that is a dialog between a user and a program. The term `@transcript` should appear on a line by itself, before the text of the dialog, and the terms `@end transcript` should appear on a line by itself, at the end of the dialog. The term `@userinput` may be used within a transcript.
- `@transport` is used to mark the names of ILU data transport systems.
- `@type` is used to mark the names of programming language types.
- `@url` is used to mark World Wide Web urls.
- `@userinput` is used to mark text typed by the user in a transcript section.

15.2 TIM Tools

ILU provides a program called `tim` to turn TIM files into either PostScript or GNU Info files. It is invoked either as

```
% tim -t INPUTFILE.tim >OUTPUTFILE.ps
```

to produce PostScript code from a .tim file, or as

```
% tim -i INPUTFILE.tim >OUTPUTFILE.info
```

to produce GNU Info code, or as

```
% tim -w INPUTFILE.tim >OUTPUTFILE.html
```

to produce World Wide Web HTML, or as

```
% tim -x INPUTFILE.tim >OUTPUTFILE.texinfo
```

to produce GNU texinfo code.

`tim` is a script written in the perl script language, so you will need to have `perl` installed to use it. See the ILU installation instructions for a location from which `perl` can be FTP'ed.

Appendix A The ILU Common Lisp Portable DEFSYSTEM Module

The ILU Common Lisp support uses files called 'sysdcl's to describe the generated lisp files for a particular interface. A *sysdcl* is similar to a UNIX 'Makefile', in that it describes the dependencies of the files of a module on each other. As part of ILU, we supply an implementation of a sysdcl interpreter, implemented in the `DEFSYSTEM` (which is also nicknamed `PDEFSYS`). The notion is that to load a module, the user loads the sysdcl which describes it, then uses the `DEFSYSTEM` commands to compile and load the files of that module. The rest of this section describes this system in more detail. All symbols described here are in the `pdefsys` package unless otherwise specified.

pdefsys:set-system-source-file (*NAME* `string`) (*PATHNAME* `pathname`) Function

Informs the defsystem utility that the definition of the system name can be found by loading the file pathname.

pdefsys:load-system-def (*NAME* (`or symbol string`) &optional (*RELOAD* `boolean t`) `=> boolean` Function

If there is a system named name and reload is false (the default), does nothing. Otherwise, loads the system defintion from a file. If `pdefsys:set-system-source-file` has been used to give an explicit source file for the system defintion, that file is used. Otherwise the file '*NAME*`-sysdcl.lisp`' is loaded from the directory specified in `pdefsys:*sysdcl-pathname-defaults*` if such a file exists. Returns false if the system was not loaded and is not already defined, true otherwise.

pdefsys:*sysdcl-pathname-defaults* Variable

Specifies the location for system declaration files. `*sysdcl-pathname-defaults*` is a list of pathnames; each location is searched for the declaration file. The default value is `(list #P"/import/commonlisp-library/sysdcl/")`.

pdefsys:defsystem (*NAME* `string`) (*SYSTEM-OPTIONS* `plist`) &rest (*MODULE-DESCRIPTIONS* `module-list`) Macro

The name of the system (which is interned in the current package), is used by defsystem to allow dependencies between multiple systems.

The *SYSTEM-OPTIONS* is a plist which may contain each of the following keywords:

- `:default-pathname` (`(or string pathname)`)

  The default place in which to find files; this value defaults to the null string. This argument is evaluated (unlike most of the others).

- `:default-binary-pathname` (`(or string pathname)`)

  The default location in which to place and look for binaries. This defaults to the value of the :default-pathname option. This argument is evaluated (unlike most of the others).

- `:default-package` (`(or symbol package)`)

  The default package to load/compile modules in; this value defaults to the current package.

- `:default-optimizations` (`list`)

  List of default compiler optimizations settings to use when compiling modules. If `nil`, optimization levels are not changed.

- `:needed-systems` (`list`)

  A list of subsystems; this value defaults to `nil`.

- `:load-before-compile` (`(or boolean list)`)

  A list of subsystems needed for compilation; this value defaults to `nil`. A value of `T` means all needed subsystems.

The module-descriptions is a list of modules which make up a system. A module is a list whose car is the module name and whose cdr is a list of keywords and values. The module keywords may contain each of the following:

- `:load-before-compile` (`list`)

  The load-before-compile keyword specifies a list of modules which will cause this module to be recompiled. If any of listed modules is newer then the current module; the current module will be recompiled. If the current module is recompiled the list of recompile dependencies will be loaded first.

  This is also a recursive recompilation. If foo dependends on bar and bar is out of date then bar will be recompiled before foo is recompiled.

  A value of `T` means all modules that occur earlier in the system definition. This value defaults to `nil`.

- `:load-after` (`list`)

  The load-after keyword specifies a list of modules which should be loaded before the current module is loaded. This option is useful only for modules during compilation since the load order will normally be satisfied during a load-system. A value of `T` means all modules that occur earlier in the system definition. This value defaults to `nil`.

- `:pathname` (`(or string pathname)`)

  The pathname keyword specifies a pathname to find the current module. Normally the pathname is the result of the concatenation of the default pathname for the system and the module name. This value defaults to `nil`. This argument is evaluated, unlike the other module options.

- `:binary-pathname` (`(or string pathname)`) Specifies the pathname for the binary of the current module. Defaults to the pathname with the same directory & name as the module source, with an appropriate file type.

- `:package` (`(or symbol package)`) The package keyword specifies a package in which to load/compile the current module. Normally the package is the default package for the system. This value defaults to `nil`.
- `:compile-satisfies-load` (`boolean`) The compile-satisfies-load keyword specifies that compiling the current module will satisfy a load (and hence the current module will not be loaded during a compile). This option is useful only for files containing macros. This value defaults to false.
- `:language` (`keyword`) The language the source is written in. See the variable `pdefsys:*language-descriptions*` for further info. The default is :LISP.
- `:optimizations` (`list`) List of compiler optimization settings to use when compiling the module. A useful value for lisp might be ((SPEED 3) (SAFETY 0)); for C ("-O"). If not present, the system's default-optimizations are used. If they too are absent, the current settings are used.
- `:libraries` (`list`) List of object libraries to load when the module is loaded. This is only useful for languages like C.
- `:features` (`list`) Run-time conditionialization, similar to #+. The module is used iff the features is "true" in the same way that #+ interpretes the features. Additionally, features may be `T` (the default) which is always true, or a list of features which is true iff at least one of the features matches.
- `:eval-after` (`form`) If present, a form that will be evaluated after the module is loaded. It should be noted that this is evaluated each time the module is loaded, whether or not the coresponding -file- is loaded.
- `:binary-only` (`boolean`) If true, declares that there is no source file associated with the module. No attempt will be name to compile it. Defaults to false.

pdefsys:*language-descriptions* Variable

An alist describing how files written in different languages are compiled and loaded. Each entry in the list is of the form (language-name source-file-type binary-file-type compile-fn load-fn). The language-name is the (keyword) name of the language. Source-file-type and binary-file-type are lists of strings; they are the file-types for source and binary files for the language. The compile-fn is symbol that will be called with three arguments to compile the source file; the pathname of the source file, the pathname of the binary output file, and a list of the optimizations declared for the module. Load-fn is a symbol that will be called with two required argument to load the binary file: the pathname of the binary, and a list of object library files to use.

The initial value of *language-descriptions* contains a description of `:lisp`, `:k&r-c` and `:ansi-c` languages. The description of `:lisp` uses the second argument to the compile-fn as a list of compiler optimization settings. The description of `:k&r-c` and `:ansi-c` uses the list as a set of additional arguments to pass to the C compiler.

pdefsys:undefsystem (*NAME* (`or symbol string`)) Macro

This macro removes the named system description from the list of all systems.

pdefsys:load-system (*NAME* (`or symbol string`)) &key (*RELOAD* `boolean nil`) (*RECURSE* `boolean nil`) (*TRACE* `boolean nil`) (*SOURCE-IF-NEWER* `boolean nil`) Function

This function loads the modules of the system with the specified name and is called recursively for all required systems. While the system is being loaded, the special variable `pdefsys:*current-system*` is bound to the name of the system.

The keyword args act as follows:

- *RELOAD*

  The reload keyword, if true, specifies that a full reload of all system modules and required systems, regardless of need. This value defaults to false.

- *RECURSE*

  If recurse is true, required systems are reloaded if the currently loaded version is not up-to-date or if the reload option is true. If recurse is false (the default), a required subsystem is not loaded if there is already a version loaded.

- *TRACE*

  If true, no module or subsystem is actually loaded. Instead a message is printed out informing you of what would have been loaded. The default value is false.

- *SOURCE-IF-NEWER*

  If true and a module's source is newer than its binary, or the binary does not exist, the source will be loaded. In all other cases, the binary will be loaded. The default value is false.

pdefsys:compile-system (*NAME* (`or string symbol`)) &key (*RECOMPILE* `boolean nil`) (*RELOAD* `boolean nil`) (*PROPAGATE* `boolean nil`) (*TRACE* `boolean nil`) (*INCLUDE-COMPONENTS* `boolean nil`) Function

This function compiles the modules of the system with the specified name and is called recursively for all required systems. While the system is being compiled, the special variable `pdefsys:*current-system*` is bound to the name of the system.

The keyword args act as follows:

- *RECOMPILE*

  The recompile keyword, if true, specifies that all modules should be recompiled, regardless of need. This value defaults to false.

- *INCLUDE-COMPONENTS*

  The include-components keyword, if true, specifies that compile-system should load all required systems. This value defaults to true.

- *RELOAD*

  The reload keyword, if true, specifies that a full reload of all system modules and required systems, regardless of need. This value defaults to false.
- *PROPAGATE*

  If true, the compile propagates to all subsystems (those required to load and to compile this system). The default is false.
- *TRACE*

  If true, no module of subsystem is actually compiled. Instead a message is printed out informing you of what would have been done. The default value is false.

pdefsys:show-system (*NAME* (`or string symbol`)) Function

This function outputs a formatted description of the system with the specified *NAME*.

A.1 Pathname Support

Some lisps don't yet support the structured directories specified in CLtL2 (p. 620). To support those lisps, `pdefsys` contains two functions which do support some of that functionality.

pdefsys:make-pathname &key *host device directory name type version defaults* Function

pdefsys:pathname-directory *pathname* Function

These functions shadow the functions in the `common-lisp` package, and support the subdirectory list syntax described as follows (From the X3J13 PATHNAME-SUBDIRECTORY-LIST proposal):

> It is impossible to write portable code that can produce a pathname in a subdirectory of a hierarchical file system. This defeats much of the purpose of the pathname abstraction.
>
> According to CLtL, only a string is a portable value for the directory component of a pathname. Thus in order to denote a subdirectory, the use of punctuation characters (such as dots, slashes, or backslashes) would be necessary. The very fact that such syntax varies from host to host means that although the representation might be "portable", the code using that representation is not portable.
>
> This problem is even worse for programs running on machines on a network that can retrieve files from multiple hosts, each using a different OS and thus different subdirectory punctuation.
>
> Proposal:
>
> Allow the value of a pathname's directory component to be a list. The car of the list is one of the symbols :ABSOLUTE or :RELATIVE. Each remaining element of the list is a string or a symbol (see below). Each string names a single level of directory structure. The strings should contain only the directory names themselves – no punctuation characters.

A list whose car is the symbol :ABSOLUTE represents a directory path starting from the root directory. The list (:ABSOLUTE) represents the root directory. The list (:ABSOLUTE "foo" "bar" "baz") represents the directory called "/foo/bar/baz" in Unix [except possibly for alphabetic case – that is the subject of a separate issue].

A list whose car is the symbol :RELATIVE represents a directory path starting from a default directory. The list (:RELATIVE) has the same meaning as `nil` and hence is not used. The list (:RELATIVE "foo" "bar") represents the directory named "bar" in the directory named "foo" in the default directory.

Here's an sample sysdcl file that shows how the DEFSYSTEM functions and these pathname functions work together.

```
(in-package "DEFSYSTEM")

(defvar *my-system-default-directory*
  (make-pathname :directory
                 '(:absolute "import" "my-system" "release-1.0")))

(set-system-source-file :mysys-test
  (make-pathname :directory '(:relative "test")
                 :name "test-sysdcl"
                 :defaults *my-system-default-directory*))

(defsystem :my-system (:default-pathname *my-system-default-directory*
                       :default-package "USER"
                       :load-before-compile ()
                       :needed-systems ())
  ...)
```

Appendix B The ILU Common Lisp Lightweight Process System

B.1 Introduction

Although it is not required by the specification, most Common Lisp implementations include a facility for multiple, independent *threads* of control (often called *lightweight processes*) within a single Lisp environment. Unfortunately, this facility is not standardized across the various implementations. Although the capabilities provided are very similar across implementations, the details of lightweight processes and the interface to them differ significantly. This situation makes it difficult to write programs that use lightweight processes and yet are portable across Common Lisp implementations.

Common Lisp ILU does not requires lightweight processes in order to function, but they are useful. In particular, servers typically make heavy use of lightweight process facilities. The purpose of the ILU CL Process Interface is to provide a standardized, portable interface to lightweight processes for use within the ILU environment. This interface isolates ILU users from the differences in the various lightweight process implementations and allows them to write programs that are portable across all implementations to which the ILU CL Process Interface has been ported. At present, these implementations include Franz Allegro CL 4.1, and Lucid Common Lisp 3.0 (a.k.a., Sun Common Lisp).

This chapter explains how the ILU CL Process Interface works for ILU users. It begins with an overview that describes the ILU CL Process model, followed by a listing of some functional capabilities of this model. After brief discussions of the implementation architecture and general limitations of the ILU CL Process Interface, the chapter presents an example of how to use the interface to define a simple shared FIFO queue. Next, it lists all of the functions and macros necessary to use lightweight processes in the ILU environment. The chapter concludes with a brief list of references.

To use the information in this chapter, you should be familiar with Common Lisp and with the notion of processes and threads in an operating system. Familiarity with the UNIX process model would also be helpful. (See the References section for recommendations on further reading.)

B.2 Overview Of The ILU CL Process Model

The ILU CL Process Interface features an interface to lightweight processes similar to that on the Symbolics Lisp machine. In particular, within a single Lisp environment (which on stock hardware runs as a single heavyweight UNIX process) there are multiple threads of control that can be scheduled independently. These threads are called *lightweight processes* (or sometimes just *processes*). Each lightweight process contains its own run-time control and binding stack, but it shares the global data and program address space with all other processes in the Lisp environment.

Note that this arrangement differs from that of the UNIX heavyweight process facility, where each process has its own address space as well as its own run-time stack.

B.2.1 The Scheduler Process

Each lightweight process represents an independent thread of control. The multiple threads within the Lisp environment are managed by a special

scheduler process. The ILU CL Process Interface makes no assumptions about the nature of this scheduler process. However, most implementations use a time-slice, priority-based scheduler. In such a scheduler, an interrupt occurs once every so often (called the scheduler's

quantum). When the interrupt occurs, the process that is currently running is stopped and its state is saved. The scheduler then examines all processes that are runnable (that is, waiting to run) and restarts the process that has the highest priority. This process runs until the next interrupt or until it gives up control to the scheduler, whichever comes first. At any given time, the one process that is "currently" running is known as the

current process.

B.2.2 States Of Processes

In the ILU CL Process model, each lightweight process is represented by a single Lisp object that maintains the information about that process. Also, each process is always in one of three states:

active, *inactive*, or *killed*. A process maintains two lists of objects called, respectively, the *run reasons* and the *arrest reasons* for the process. For a process to be active, it must have at least one run reason and no arrest reasons. A process with no run reasons or at least one arrest reason is considered inactive. The ILU CL Process Interface provides functions for adding and removing run and arrest reasons for a process. Thus, the user (or a program) can move a process between the active and inactive states.

The scheduler runs only active processes. Until an inactive process is reactivated, it cannot run. A killed process is one that has been explicitly killed (using the `ilu-process:process-kill` function). A killed process can never be run again (that is, it can never be made active).

An active process can in turn be in one of two substates: *runnable* and *waiting*. A runnable process is ready to be restarted by the scheduler, which determines whether and when a process will actually be restarted based on its status (that is, priority) and the status of the other runnable processes. A waiting process is a process that has a *wait function* and a list of *wait arguments*. These two items are supplied to the process using the `ilu-process:process-wait` function. Periodically, the scheduler will `apply` the process's wait function to its wait arguments (in the context of the scheduler). If the result is a non-`nil` value, the wait function and wait arguments are removed from the process, and the process thereby becomes runnable. Usually, the scheduler evaluates the wait functions for all waiting processes every time around the scheduler loop. Therefore, it is important that wait functions be fast and very efficient.

B.2.3 Removing Or Killing Processes

You can reversably remove a process from a runnable state either by entering a wait or by making it inactive. In general, it is more efficient to make a process inactive because this removes it from the scheduler's active process list. Thus, the scheduler does not incur the cost of periodically evaluating its wait function. However, an inactive process cannot make itself active. It must depend on some other process to recognize when it is ready to run again and to reactivate it at that time. Although a waiting process is initially more costly than an inactive one, it is automatically returned to a runnable state by the scheduler whenever its wait function returns non-`nil`. Hence, no second process is needed to *restart* a waiting function. Thus, the choice between waiting a process and rendering it inactive depends on the architecture of the application being written.

When a process is first started, it is given a Lisp function and a set of arguments to this function. These are known as the process's

initial-function and *inital-arguments*, respectively. A newly created process, `apply`s its initial-function to its inital-arguments. When the initial-function returns, the process is automatically killed. Once killed it can never be restarted. You can also kill the process before the inital-function returns using the `ilu-process:process-kill` function, which causes the process to execute a `throw` in its current context. This `throw` causes the stack to unwind (executing unwind-protect forms along the way) and the initial-function to return, thereby killing the process.

B.2.4 Properties Of Processes

Every process has a number of properties. Specifically, a process has an arbitrary *process name* that identifies it in displays and in certain operations. Process names need not be unique. A process also has a

priority that the scheduler uses optionally to determine when to schedule the process. Priorities are small integers and default to zero (0). In most implementations, processes with higher priorities are given scheduling preference. Negatives are used to indicate that a process should run as a background task when nothing else is running. Finally, a process has a *quantum*, which is the amount of time (measured in seconds) that the process wishes to run each time before it is interrupted. In some implementations, the scheduler uses a process's quantum to help determine the actual length of the time-slice given to the process. Many implementations ignore the quantum altogether.

B.2.5 Process Locks

The ILU CL Process Interface also includes a facility called *process locks* that supports exclusion-based sharing of a common resource (that is, a common object or data structure) or a critical region of code by two or more concurrent processes. A process lock is an object that a process can *lock* in order to claim exclusive access to the shared resource corresponding to the lock. Process locks are essentially a semaphore mechanism specialized for use with the ILU CL Process interface.

Each process lock has a name and a locker. A lock's name is for display purposes only. Processes can ask to gain or relinquish exclusive rights to the lock (called *locking* and *unlocking* the lock, respectively). While a process has rights to the lock, the lock's locker is (generally) the process object for that process. When a process asks to lock a lock that is already locked, the asking process blocks and "waits" until the lock is free. Waiting does not necessarily use the standard wait mechanism. Some implementations use process deactivation to implement the "wait" in this case. Some implementations may also maintain a queue of processes waiting for a lock to be freed, thereby ensuring fair access to the lock. Other implementations may not maintain such a queue, and therefore fair access to the lock is not guaranteed.

Process locks are contractual in nature. The various processes sharing a resource (or critical section of code) must all agree not to access the common resource while the process lock corresponding to that resource is held by another process. Furthermore, they must agree to lock the process lock whenever they need exclusive access to the resource, thereby notifying the other processes of their intent. Moreover, the correspondence between a process lock and the shared resource is a matter of agreement between the cooperating processes. The system does not provide any direct support for this correspondence (although it may be added on at a higher level built on top of the basic process lock mechanism).

Process locks provide a code-centered "sharing" mechanism where the access control is built into the programs that access the shared resource. Process locks are suited for closed, or non-extensible, applications where the shared resource is a standard Lisp data structure (that is, not a CLOS object) and where efficiency is a major concern. For applications not meeting these criteria, a mechanism in which a CLOS object itself controls simultaneous access to its internal data structures may be more appropriate.

B.3 Functional Overview

The ILU CL Process Interface provides all of the functions and macros necessary to use lightweight processes in the ILU environment. The functionality provided by these functions and macros includes:

- Starting new processes and killing processes
- Displaying status information, such as the current process, all active processes, or all known processes
- Accessing and modifying the properties of a process (for example, its name or priority)
- Adding/Removing arrest and run reasons for a process

- Allowing a process to give up control to the scheduler or enter into a wait state
- Temporarily turning off the scheduler so that the current process cannot be interrupted
- Creating, locking, unlocking, and modifying process locks

B.4 Implementation Architecture

The ILU CL Process Interface is implemented as a veneer over the existing process interfaces for a number of Common Lisp implementations (currently Franz Allegro CL and Lucid Common Lisp). In many cases, the implementation's functions are simply imported and then exported from the `ilu-process` package. In other cases, a new function is wrapped around the implementation's native function to change the name, arguments, or semantics of the function so that they match those required by the ILU CL Process Interface specification. In a few cases, whole new functions have been written to achieve functionality not provided by the original implementation.

The nature of the process object in the ILU CL Process Interface is not specified. The process object is inherited from the underlying implementation and may therefore be a list, a structure, a flavor object, or even a CLOS object. Because of this lack of specification, process objects cannot be specialized. Moreover, they cannot be accessed or modified in any way other than through the functional interface described in this chapter.

B.5 General Limitations

The ILU CL Process Interface assumes that the scheduler is loaded and running in the ILU environment. Procedures for starting the scheduler are not included in the ILU CL Process Interface. Some implementations, however, may require you to actually load and start up the scheduler. For example, in Franz Allegro CL, you need to evaluate `(mp:start-scheduler)` either at the top-level or in your '`.clinit.cl`' file in order to load and start up the scheduler.

The ILU CL Process Interface is subject to all of the limitations of its underlying implementations. In particular, one problem with most Common Lisp implementations on stock hardware is that the smallest scheduler quantum possible is one second. This means that each process gets to run for one second uninterrupted. For applications that involve real-time response, waiting for one second before an event can be handled is problematic. In practice, this problem can be lessened if all processes release control to the scheduler at regular, short intervals (that is, each few times around a tight inner loop), thereby making the effective quantum significantly less than one second. Note that this practice effectively reduces the scheduler to a prioritized, cooperative scheduler rather than the preemptive scheduler intended.

Most Common Lisp implementations build their process mechanism on top of a very powerful mechanism called *stack groups*. Stack groups provide for alternative run-time stacks in the Lisp environment that can be used for various purposes beyond implementing processes. For example, stack groups are an ideal substrate for implementing co-routines. Unfortunately, not all implementations provide an interface to stack groups (if indeed they have stack groups). Hence, an interface to stack groups is not a part of the ILU CL Process Interface.

B.6 How To Use The ILU CL Process Interface

The ILU CL Process Interface is intended as a programmer's interface; the functions and macros provided should be used to implement programs that run in the ILU environment. Although you can use any of the functions and macros directly from a Lisp listener, the interface is not designed particularly well for interactive use. The two exceptions to this rule are the functions `ilu-process:show-process` and `ilu-process:show-all-processes`, both of which are designed to print out status information in the Lisp listener window. Because it is a user-oriented function, `ilu-process:show-process` accepts either the process name or a process object to identify the process whose status is to be displayed.

Most implementations include an interactive interface to multiple processes and the scheduler. For example, Franz Allegro CL has a special top-level command language that is operative in every Lisp listener. This command language includes the following commands that deal specifically with lightweight processes (see Chapter 4 of *[Franz-92]* for more information):

- `:processes`

  Lists all processes (see `ilu-process:all-processes`)
- `:kill`

  Kills a process (see `ilu-process:process-kill`)
- `:arrest`

  Adds an arrest reason to a process (see `ilu-process:add-arrest-reason`)
- `:unarrest`

  Removes any arrest reason that was added to a process by `:arrest` (see `ilu-process:process-revoke-arrest-reason`)
- `:focus`

  Performs an `:arrest` on a process and arranges for all user keyboard input to be sent to the arrested process (usually to the debugger).

B.7 How To Program The ILU CL Process Interface

The following example illustrates how to use the ILU CL Process Interface to define a shared FIFO queue. Two processes will utilize this queue. A producer process will read input items from the user and place them on the shared queue. A consumer process will wake up every five seconds and read items from the shared queue, printing them on the standard output stream as they are taken off the queue. Access to the shared queue will be controlled using a process lock associated with the queue.

```
;;;-----------------------------------------------
;;; the shared queue, its process-lock, and its accessors/mutators
;;;

(defvar queue (list t) "The shared queue")

(defvar queue-lock (ilu-process:make-process-lock :name "queue lock")
  "process lock for queue")

(defun queue-pop (queue)
  "Pop an item off of the shared FIFO queue.
   Use ilu-process:with-process-lock to prevent collisions between processes.
  "
  (ilu-process:with-process-lock (queue-lock)
    (prog1
(cadr queue)
      (rplacd queue (cddr queue)))
    ))

(defun queue-push (queue item)
  "Push an item onto the shared FIFO queue.
   Use ilu-process:with-process-lock to prevent collisions between processes.
  "
  (ilu-process:with-process-lock (queue-lock)
    (nconc queue (list item))
    ))

(defun queue-empty-p (queue)
  "Is queue empty?
   Use ilu-process:with-process-lock to prevent collisions between processes.
  "
  (ilu-process:with-process-lock (queue-lock) (null (cdr queue)) ))

;;;-----------------------------------------------
;;;  The producer function
;;;

(defun produce ()
  "Loop reading an item from the user and pushing it onto the shared queue."
  (let (Item)
    (loop
      ;; Wait until there is something on the input stream.
      (ilu-process:process-wait "Waiting for input" #'listen *standard-input*)

      ;; Read the input.
      (setq Item (read *standard-input*))

      ;; Check to see if it is the EOF marker and exit if so.
      (when (eq Item :EOF) (return nil))
```

```
      ;; Push the item onto the queue.
      (queue-push queue Item)
      )))

;;;-----------------------------------------------
;;; The consumer function
;;;

(defun consume ()
  "Wake up every five seconds and see if there is something on the shared
   queue.  If there is, pop it off and print it on standard output.
   If the queue is empty and the producer process is not alive, terminate.
  "
  (loop
    ;; Check to see if there is anything on the queue.
    (if (not (queue-empty-p queue))
;; There is an item on the queue; pop and print all items.
(do ()((queue-empty-p queue))
  (fresh-line t)
  (princ "Output: ")
  (prin1 (queue-pop queue))
  (fresh-line t)
  (finish-output t))

      ;; Queue is empty; check to see if the producer is still alive.
      (if (null (ilu-process:find-process "Producer Process"))

  ;; Producer not alive; terminate.
  (return nil)))

    ;; Sleep for five seconds; this gives up control immediately
    ;; so some other process can run.
    (sleep 5)
    ))

;;;-----------------------------------------------
;;;   Main function; starts consumer and producer processes
;;;

(defun test-queue ()
  "Start consumer and producer processes. Wait in an idle loop until
   both the producer and the consumer processes die.  This function is
   meant to be evaluated in the Lisp listener.  Waiting until both
   processes die ensures that the Lisp listener does not interfere
   with user input to the producer.
  "
  (let (Producer Consumer)
    ;; Start the producer first; the consumer needs the producer to run.
    (setq Producer (ilu-process:fork-process "Producer Process" #'produce))
    ;; Start the consumer.
```

```
    (setq Consumer (ilu-process:fork-process "Consumer Process" #'consume))
    ;; Show processes on the standard ouput.
    (ilu-process:show-all-processes)
    ;; Wait until both consumer and producer are dead.
    (ilu-process:process-wait "Waiting for godot"
#'(lambda (P1 P2)
    (not
     (or (ilu-process:process-alive-p P1)
 (ilu-process:process-alive-p P2))))
Consumer Producer)
    ))
```

The following is a transcript of this test program in operation:

```
;;;--------------------------------------------------
#73: (test-queue)
-------------Data on all processes follows---------
Process: "Consumer Process"
  Process-alive-p: T
  Process-active-p: T
  Process-quantum: 2
  Process-priority: 0
  Process-run-reasons: (:START :START)
Process: "Producer Process"
  Process-alive-p: T
  Process-active-p: T
  Process-quantum: 2
  Process-priority: 0
  Process-run-reasons: (:START :START)
Process: "Initial Lisp Listener"
  Process-alive-p: T
  Process-active-p: T
  Process-quantum: 2
  Process-priority: 0
  Process-run-reasons: (:ENABLE)
123
Output: 123
456
789
Output: 456
Output: 789
444
555
666
Output: 444
Output: 555
Output: 666
:eof
NIL
#74:
```

B.8 The ILU CL Process Interface

The following sections detail the functions and macros that make up the ILU CL Process Interface. All are assumed to be in the `ilu-process` package unless otherwise specified. Arguments are shown with their type, if they have any restrictions on their type. Return types are shown if the function returns a value. Optional arguments are shown with their type and their default value.

B.8.1 The Process Object

The following listings describe the object that is used to represent each lightweight process.

ilu-process:process Type

A Lisp object representing a single process. This object is to be used only as a handle for the process. To alter the state or characteristics of a process, use the external function interface defined below. The exact nature of the process object differs between implementations. In particular, it may or may not be a flavor or a CLOS object. Hence, it is not safe to specialize processes.

find-process (*NAME* `string`) `=> process` Function

Returns the process object whose name is *NAME*. Only `ilu-process:process-alive-p` processes (that is, processes on the list returned from `ilu-process:all-processes`) are searched. This function returns `nil` if there is no matching process.

processp *OBJECT* `=> boolean` Function

Returns non-`nil` if *OBJECT* is an object of type `process` for this implementation. This function returns `nil` otherwise.

B.8.2 Querying The Status Of The Scheduler And All Processes

The following functions and macros provide status information about the general state of processes and the scheduler in the Lisp environment.

active-processes `=> list` Macro

Returns a list of all active processes; that is, processes that have at least one run reason and no arrest reasons. Note, however, that these processes are not necessarily runnable because they may be in a process-wait.

all-processes `=> list` Macro

Returns a list of all processes currently known by the scheduler, including active and inactive processes but not processes that have been killed.

current-process `=> process` Macro

Returns the process object for the current thread of control.

show-all-processes &optional (*STREAM* `streamp` `cl:*standard-output*`) (*VERBOSE* `boolean` `nil`) — Function

Displays information about all processes known by the scheduler (that is, the processes returned by `ilu-process:all-processes`). Output is to *STREAM*, which defaults to the value of `cl:*standard-output*`. This function shows only non-`nil` fields unless *VERBOSE* is non-`nil`; the default is `nil`.

B.8.3 Starting And Killing Processes

fork-process (*NAME-OR-KEY-LIST* `(or string proplist)`) (*FUNCTION* `function`) &rest *ARGS* `=> process` — Function

Creates a new process and returns the `process` object for this process. In this process, *FUNCTION* is applyed to *ARGS*. If *FUNCTION* ever returns, the process is automatically killed. The *FUNCTION* is known as the initial-function of the process (see `ilu-process:process-initial-form`).

The new process is activated by default, although you can create it in a deactivated state by giving it a run reasons list with a value of `nil` or by giving it one or more arrest reasons as detailed below.

NAME-OR-KEY-LIST is either a string, in which case it serves as the name of the process, or it is a property list with one or more of the following property-value pairs:

- `:name` (`string`)

  A string to be used as the name of the process.
- `:priority` (`integer`)

  Sets the priority of the process to the given value (see `ilu-process:process-priority`).
- `:quantum` (`(or numberp nil)`)

  Sets the quantum of the process to the given value (see `ilu-process:process-quantum`). Defaults to `1`.
- `:stack-size` (`fixnum`)

  Sets the stack-size of the process (if possible in this implementation).
- `:run-reasons` (`list`)

  Sets the run reasons of this process to the given list. Unless run-reasons is non-`nil`, the forked process does not run until a `ilu-process:process-add-run-reason` is done. This property defaults to `(quote (:start))`.
- `:arrest-reasons` (`list`)

  Sets the arrest reasons of this process to the given list. If arrest-reasons is non-`nil`, the forked process does not run until a `ilu-process:process-revoke-arrest-reason` is done. This property defaults to `nil`.

- `:bindings` (`list`)

  A list of bindings (as in let) that are done in the context of the forked process before the function is run. This property defaults to `ilu-process:*default-process-bindings*`.

process-kill (*PROCESS* `process`) Function

Terminates *PROCESS* and removes it from the scheduler's consideration altogether. It is an error if *PROCESS* is not `ilu-process:processp` and `ilu-process:process-alive-p`.

A process may not terminate immediately. In particular, the process is first activated and scheduled. It is then forced to `throw` out of its initial-function, thereby properly unwinding and executing any unwind forms.

A killed process cannot be reactivated.

B.8.4 Waiting A Process

process-wait (*WHOSTATE* `string`) (*FUNCTION* `function`) &rest *ARGS* Function

The current process is suspended until *FUNCTION* applied to *ARGS* returns non-`nil`. During this time, the process's whostate (see `ilu-process:process-whostate`) is set to *WHOSTATE*.

Note that the current process is not deactivated. It is simply not scheduled to run until its wait-function returns non-`nil`. The scheduler re-evaluates the wait-function periodically. In general, the re-evaluation occurs whenever the waited process would be scheduled to run if it were not suspended. However, in some implementations it is run during every scheduler break.

B.8.5 Activating And Deactivating Processes

process-add-arrest-reason (*PROCESS* `process`) *OBJECT* Function

Adds *OBJECT* to the list of arrest reasons for *PROCESS*. The *OBJECT* argument can be any Lisp object. It is an error if *PROCESS* is not `ilu-process:processp`.

Adding an arrest reason may cause a process to become deactivated. In particular, if this is the first arrest reason, then the process becomes deactivated (if it was previously activated).

process-add-run-reason (*PROCESS* `process`) *OBJECT* Function

Adds *OBJECT* to the list of run reasons for *PROCESS*. The *OBJECT* argument can be any Lisp object. It is an error if *PROCESS* is not `ilu-process:processp`.

Adding a run reason may cause a process to become activated. In particular, if there are no arrest reasons and the added run reason is first, the process goes from a deactivated state to an activated state.

process-arrest-reasons (*PROCESS* `process`) `=> list` Function

Returns the list of arrest reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`.

process-disable (*PROCESS* `process`) Function

Causes *PROCESS* to become inactive by removing all of its arrest reasons and all of its run reasons. It is an error if *PROCESS* is not `ilu-process:processp`.

process-enable (*PROCESS* `process`) Function

Causes *PROCESS* to become active by removing all of its arrest reasons and all of its run reasons and then giving it a single run reason (usually `:enable`). It is an error if *PROCESS* is not `ilu-process:processp`.

process-revoke-arrest-reason (*PROCESS* `process`) *OBJECT* Function

Removes *OBJECT* from the list of arrest reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. *OBJECT* is compared to the existing arrest reasons using an `eq` test.

Revoking an arrest reason may cause a process to become activated. In particular, when the last arrest reason for a process is removed, the process is (re-)activated if it has at least one run reason.

process-revoke-run-reason (*PROCESS* `process`) *OBJECT* Function

Removes *OBJECT* from the list of run reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The *OBJECT* argument is compared to the existing run reasons using an `eq` test.

Revoking a run reason may cause a process to become inactive. In particular, when the last run reason for a process is removed, the process is made inactive (if it was previously activate).

process-run-reasons (*PROCESS* `process`) Function

Returns the list of run reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`.

B.8.6 Accessing And Modifying The Properties Of A Process

process-active-p (*PROCESS* `process`) `=> boolean` Function

Returns non-`nil` if *PROCESS* is an active process object; that is, a process with no arrest reasons and at least one run reason. Otherwise, this function returns `nil`. It is an error if *PROCESS* is not `ilu-process:processp`.

process-alive-p (*PROCESS* `process`) `=> boolean` Function

Returns non-`nil` if *PROCESS* is alive (that is, has been created but has not been killed). Essentially, a process is alive if it is on the list returned by `ilu-process:all-processes`. It is an error if *PROCESS* is not `ilu-process:processp`.

process-initial-form (*PROCESS* `process`) `=> consp` Function

Returns the initial-form of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. Note that the returned value is not an `eval`able form. It is merely the `cons` of the process's intial function onto a list of the initial arguments passed to the function. (See `ilu-process:fork-process`.)

process-name (*PROCESS* `process`) `=> string` Function

Returns the name of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-active-p` function can be used with `setf` to change the name of a process.

process-priority (*PROCESS* `process`) `=> integer` Function

Returns the scheduling priority for the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-priority` function can be used with `setf` to change the priority of a process.

When the priorities are set, a small integer is generally used. Process priorities default to zero (0). Processes with higher priorities are given scheduling preference. Priorities can be negative if a process should run as a background task when nothing else is running.

Note that an implementation is free to ignore process priorities. Setting a process's priority is merely advisory. For this reason, the value returned by `ilu-process:process-priority` may not match the most recent `setf` on `ilu-process:process-priority`.

process-quantum (*PROCESS* `process`) `=> (or numberp nil)` Function

Returns the quantum, which is the amount of time the scheduler allows a process to run each time its is rescheduled, for the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-quantum` function can be used with `setf` to change the quantum of a process.

The quantum is measured in seconds (not necessarily integral).

Note that an implementation is free to ignore process quantums. Setting a quantum is merely advisory. For this reason, the value returned by `ilu-process:process-quantum` may not match the most recent `setf` on `ilu-process:process-quantum`.

The default process quantum is 1.

process-wait-args (*PROCESS* `process`) `=> list` Function

Returns a list of the arguments being passed to the wait-function of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. (See `ilu-process:process-wait`.)

process-wait-function (*PROCESS* `process`) `=> (or functionp nil)` Function

Returns the wait-function of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. (See `ilu-process:process-wait`.)

process-whostate (*PROCESS* `process`) `=> string` Function

Returns the whostate of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-whostate` function can be used with `setf` to change the whostate of a process. (See also `ilu-process:fork-process` and `ilu-process:process-wait`.)

show-process &optional (*PROCESS* `process`) (*STREAM* `streamp` `cl:*standard-output*`) (*VERBOSE* `boolean nil`) Function

Displays information about process *PROCESS*, which may be a `process` object or the name of a process known to the scheduler. If *PROCESS* is a symbol, it is downcased and converted to a string. *PROCESS* defaults to the current process. Output is is to *STREAM*, which defaults to the value of `cl:*standard-output*`. If *VERBOSE* is `nil` (defaults to non-`nil`), then only non-`nil` fields are displayed and the process's initial-form is not shown.

B.8.7 Miscellaneous Process/Scheduler Functions And Macros

process-allow-schedule Function

Suspends the current process and returns to the scheduler. All other processes of equal or higher priority have a chance to run before control returns to the current process.

process-interrupt (*PROCESS* `process`) (*FUNCTION* `function`) &rest *ARGS* Function

Forces *PROCESS* to apply *FUNCTION* to *ARGS* when it is next scheduled. When *FUNCTION* returns, *PROCESS* resumes execution where it was interrupted.

In general, `ilu-process:process-interrupt` is run immediately (that is, when *PROCESS* is next scheduled) if *PROCESS* is active, even if *PROCESS* is a process-wait. If

PROCESS is not active, `ilu-process:process-interrupt` may wait until *PROCESS* is reactivated before *FUNCTION* is executed.

without-scheduling &body *BODY* Macro

Evaluates the forms in *BODY* with scheduling turned off. While the current-process is within the scope of `ilu-process:without-scheduling`, no other process will run. However, scheduling may be resumed if a `ilu-process:process-wait` or `ilu-process:process-allow-schedule` is executed within the scope of `ilu-process:without-scheduling`. Most Common Lisp I/O functions as well as the function `sleep` usually call some form of `ilu-process:process-allow-scheduling` and hence will resume scheduling if called within the scope of a `ilu-process:without-scheduling`.

B.8.8 Process Locks Interface

ilu-process:process-lock Type

The process lock object. You should access fields of this lock using only the functional interface listed in this section.

make-process-lock &key (*NAME* `(or nil string) nil`) `=> process-lock` Function

Creates and returns a `process-lock` object with *NAME* as the name of the lock.

process-lock (*LOCK* `ilu-process:process-lock`) &optional (*LOCK-VALUE* `process (ilu-process:current-process)`) (*WHOSTATE* `(or nil string) <undefined>`) Function

Grabs *LOCK*, entering *LOCK-VALUE* as the lock's locker. *LOCK-VALUE* defaults to the current process. It is an error if *LOCK* is not `ilu-process:process-lock-p`.

If *LOCK* is already locked, then the current process waits until it is unlocked. The waiting is done using `ilu-process:process-wait`. The *WHOSTATE* argument is a string that is used as the whostate of the process if the process is forced to wait; defaults to an implementation-dependent string.

process-lock-locker (*LOCK* `ilu-process:process-lock`) `=> t` Function

Returns the current locker of *LOCK*. It is an error if *LOCK* is not `ilu-process:process-lock-p`. This function returns `nil` if *LOCK* is currently unlocked, that is, has no locker. This value is *not* setfable. You should use `ilu-process:process-lock` to set the locker.

process-lock-name (*LOCK* `ilu-process:process-lock`) => (`or nil string`) — Setf-able Function

Returns the name associated with *LOCK*. It is an error if *LOCK* is not `ilu-process:process-lock-p`. The `ilu-process:process-lock-locker` function can be used with `setf` to change the name of a process lock.

process-lock-p *OBJECT* `=> boolean` — Function

Returns non-`nil` if *OBJECT* is a `ilu-process:process-lock`. Otherwise, this function returns `nil`.

process-unlock (*LOCK* `ilu-process:process-lock`) &optional (*LOCK-VALUE* `t (ilu-process:current-process)`) (*ERROR-P* `boolean nil`) — Function

Releases *LOCK*. It is an error if *LOCK* is not `ilu-process:process-lock-p`.

If *LOCK*'s locker is not `eq` to *LOCK-VALUE*, which defaults to the current process, then an error is signalled unless *ERROR-P* is `nil` (it defaults to `t`).

with-process-lock (*LOCK* `ilu-process:process-lock`) &key (*NORECURSIVE* `boolean nil`) &body *BODY* — Macro

Locks *LOCK* for the current process and evaluates the forms in *BODY*. It is an error if *LOCK* is not `ilu-process:process-lock-p`.

If *NORECURSIVE* is `t` (the default), and if the current process already owns the lock (determined dynamically), then no action is taken. If *NORECURSIVE* is non-`nil`, then an error is signalled if the current process already owns *LOCK*.

If *LOCK* is held by another process, then the current process waits as in `ilu-process:process-lock`, which is described earlier in this section.

B.9 Handling Errors

Errors in most of the process functions will cause a break. There are no special tricks to handling these errors.

The `:focus` command is an important tool for using the Allegro CL debugger in a multiple-process environment. In particular, in Allegro CL a new process by default shares its standard input/output (I/O) stream with the Lisp listener. Generally, this is not a problem because the process runs in the background and does no I/O. However, if the process enters a break, the debugger needs to use the process's standard I/O stream to interact with the user. This could lead to a problem because the debugger I/O from the broken process will be interleaved with the Lisp listener's normal I/O. Specifically, the system will not be able to determine to which process user input is directed.

To avoid this situation, Allegro CL has the notion of a *focus process*. Input coming from the shared Lisp listener I/O stream is always sent to the focused process. Usually this is the Lisp listener process. However, if a background process breaks, you can use the `:focus` command to focus on the broken process and allow you to send input to the debugger running in that process. When the debugging is complete, `:focus` is automatically returned to the Lisp listener process.

The following transcript illustrates the use of the `:focus` command in Allegro CL:

```
;;;-----------------------------------------------------------------------
;;; Start out focused on the Lisp listener process. List all processes.
<cl 71> :processes
"Initial Lisp Listener" is active.
;;;-----------------------------------------------------------------------
;;; Second, start a test process that will enter a break immediately.
<cl 72> (ilu-process:fork-process "test" #'error "test break")
#<process test  #x13e92b1>
<cl 73>
;;;--------------------------------------------------
;;; Process test enters a break.

Error: test break

;;; Still speaking to the Lisp listener process, list the processes.
[1] <cl 1> :processes
"test" is waiting for terminal input.
"Initial Lisp Listener" is active.
<cl 74>
;;;--------------------------------------------------
;;; Now refocus on the test process.
<cl 74> :focus "test"
Focus is now on the "test" process.
;;;  Look at stack of test process.
<cl 75> :zoom
Evaluation stack of process "test":
 ->(EXCL::STM-SY-READ-CHAR #<synonym stream for *TERMINAL-IO*  #x13e9619>)
   (PEEK-CHAR NIL #<synonym stream for *TERMINAL-IO*  #x13e9619> NIL :EOF NIL)
   (ERROR "test break")
   (ILU-PROCESS::PROCESS-INITIALIZATION-FUNCTION NIL
      #<Function ERROR  #x219ab9> ("test break"))
;;;--------------------------------------------------
;;; Kill the test process (which is the current process).
<cl 76> :kill
Do you really want to kill process "test" [n]? y

;;; Automatic refocus to Lisp listener. Ask listener to list all processes.
Focus is now on the "listener" process.
<cl 77> :processes
"Initial Lisp Listener" is active.
```

For more information on the Lisp listener interface and the Lisp debugger, see the manual for the implementation of Common Lisp that you are using. For Allegro CL, refer to chapters 4 and 5 of *[Franz-92]*.

B.10 Notes

It is possible for a process to do a non-blocking attempt to lock a process lock using the following idiom:

```
(ilu-process:without-scheduling    ; Make sure this is not interrupted.
 (if (ilu-process:process-lock-locker LOCK) ; Is lock free?
     (ilu-process:process-lock LOCK)))      ; Lock is free, grab it.
(if (eq                            ; Did we get the lock for this process?
     (ilu-process:process-lock-locker LOCK)
     (ilu-process:current-process))
    (prog1                         ; Yes, do A, releasing lock on way out.
      ...A...
      (ilu-process:process-unlock LOCK))
  ...B...                          ; No, do B, which does not depend on lock.
  )
```

B.11 References

[Franz-92]: **Allegro Common Lisp User Guide**. Release 4.1. Berkeley, CA: Franz Incorporated, March 1992.

Bach, M.J., **The Design of the UNIX Operating System**. Englewood Cliffs, NJ: Prentice-Hall, 1986. Especially read Chapters 6, 7, and 8.

Deitel, H.M. **An Introduction to Operating Systems**. Reading, MA: Addison-Wesley, 1984. Especially read Part 2.

Kernighan, B.W. and R. Pike., **The UNIX Programming Environment**. Englewood Cliffs, NJ: Prentice-Hall, 1984. Especially read Sections 1.4 and 7.4.

Tanenbaum, A.S. **Operating Systems: Design and Implementation**. Englewood Cliffs, NJ: Prentice-Hall, 1987. Especially read Chapter 2.

Appendix C Porting ILU to Common Lisp Implementations

The ILU runtime for Common Lisp is largely written in vanilla Common Lisp. The lisp-implementation-specific details are confined to a small number of macros and functions which need to be defined. (This assumes that you have a working port of ILU and its C support already on your operating system platform. If not, you will have to begin by doing that.) Aside from these macros and functions, you do not require anything not specified in the Common Lisp standard. You do not need Lisp code for TCP/IP or socket support. The major work is to write ilu-xxx.lisp, where "xxx" is the specifier for the particular implementation of Common Lisp in use, and any necessary xxx-to-C shims in ilu-xxx-skin.c. There are a number of things that have to be done in ilu-xxx.lisp. They can be regarded in three major sections: providing the ILU notion of foreign-function calls, connecting the Lisp's garbage collector to the ILU network GC, and providing either a threaded or event-loop model of operation. In addition, there is a small hook that has to be provided to convert between character sets.

C.1 Providing the ILU notion of foreign-function calls.

Perhaps the trickiest is to provide an implementation of the macro "define-c-function". This maps the ILU notion of a call into C into the native lisp notion. "define-c-function" has the signature

ilu::define-c-function (*LISP-NAME* `symbol`) (*DOC-STRING* `string`) (*C-NAME* `string`) (*ARGS* `list`) (*RETURN-TYPE* `keyword`) &key (*INLINE* `boolean cl:nil`) — Macro

The *LISP-NAME* is a symbol which will be the name of the function in Common Lisp. The *C-NAME* is a string which will be the “regular” C name of the C function to be called; that is, the name as it would be named in a C program, rather than the name of the symbol for the entry point of the function. *ARGS* is a list of arg which describe the signature of the C function, where each arg is either a keyword or a 2-tuple. If a keyword, the keyword indicates the type of the argument. Allowable argument types are

- `:short-cardinal` (unsigned-byte 16)
- `:cardinal` (unsigned-byte 32)
- `:short-integer` (signed-byte 16)
- `:integer` (signed-byte 32)
- `:short-real` (single-float)
- `:real` (double-float)
- `:byte` (0 <= fixnum < 256)
- `:boolean` (t or nil)

- `:fixnum` (-2^27 <fixnum < 2^27 (about))
- `:string` (string)
- `:constant-string` (string)
- `:bytes (vector of (unsigned-byte 8))`
- `:unicode` (Unicode if your Lisp supports it, vector of (unsigned-byte 16) otherwise)
- `:ilu-call` (unsigned-byte 32)
- `:ilu-object` (unsigned-byte 32)
- `:ilu-class` (unsigned-byte 32)
- `:ilu-server` (unsigned-byte 32)
- `:char*` (unsigned-byte 32)
- `:pointer` (unsigned-byte 32)

If the arg is a 2-tuple, the cadr is the type, and the car is the "direction", which may be either :in, `:out`, or `:inout`. Args with no "direction" are by default of direction `:in`. The *RETURN-TYPE* argument is a keyword for the return type of the function, which is drawn from the same set of keywords as the argument types. Return-types may also use the keyword :void, which specifies that no value is returned. The *INLINE* keyword is a boolean value which, if `cl:t`, indicates that the necessary type-checking has been assured by the application code, and that the C function may be called directly without type-checking the parameters.

`define-c-function` defines a Common Lisp function with a possibly different signature from the C function. This function has arguments which consist of all the `:in` and `:inout` arguments of the C function, in the order in which they occur in the signature of the C function. It returns possibly multiple values, which consist of the specified return type, if not `:void`, followed by any `:out` and `:inout` arguments to the C function, in the order in which they occur in the signature of the C function.

`define-c-function` assumes that the C function will call back into Common Lisp, and that gc may occur during the invocation of the C function. Therefore, any objects passed to C which are not values must be registered in some way to prevent them from moving during the call. Often this means that the actual call must be surrounded by code which makes static copies of, for example, strings, calls the C function, then frees the static copy after the call. In addition, when "catching" `:out` arguments and `:inout` arguments, it is usually necessary to pass a pointer to the appropriate argument, rather than the argument directly. Typically 1-element arrays have to be allocated to do this. The Franz ACL implementation uses a resource of arrays to minimize consing for this.

We should probably add another keyword, *NO-CALLBACKS*, to indicate that the C function will not call back into Common Lisp (and therefore some of the GC protection can be skipped when calling this function). Providing for *NO-CALLBACKS* in your implementation would probably be a good idea.

C.2 Network Garbage Collection

The Common Lisp-specific runtime must provide three calls which allow the kernel to map the kernel's C ILU object to a CLOS object. These are register-lisp-object, lookup-registered-lisp-object, and unregister-lisp-object. The idea behind these is to provide the C runtime with a handle on a CLOS object that is a small integer that will not be moved by Common Lisp GC, and to provide a layer which weak references can hide behind.

ilu::register-lisp-object (*OBJ* `ilu:ilu-object`) &key (*REFTYPE* `keyword :strong`) `=> fixnum` — Function

The OBJ is an ILU CLOS object (the Franz ACL implementation accepts any Common Lisp value except NIL, but this is only because it uses it internally in '`ilu-franz.lisp`'). The *REFTYPE* keyword may be either the keyword `:weak` or the keyword `:strong`, which determines whether the reference to the object is a weak reference or a strong reference. A weak reference is one that is not "followed" by the Common Lisp collector. The returned value is a fixnum that can be used with `lookup-registered-lisp-object` and `unregister-lisp-object` to find the object or remove the reference to the object, respectively.

ilu::lookup-registered-lisp-object (*INDEX* `fixnum`) `=> ilu:ilu-object` — Function

This function follows the reference indicated by *INDEX* and returns the object, or `cl:nil` if the *INDEX* is invalid.

ilu::unregister-lisp-object (*INDEX* `fixnum`) — Function

Causes any reference indicated by *INDEX* to be removed.

ilu::optional-finalization-hook (*OBJ* `ilu:ilu-object`) — Macro

This is a macro which should be defined in such a way as to indicate a finalization action for *OBJ* when the Common Lisp collector collects it. This finalization action will interact with the ILU kernel to ensure that remote peers of this Common Lisp will know that it no longer has an interest in the object. In addition, the finalization action will be able to prevent *OBJ* from being actually collected, should any peer have an active reference to it.

The Franz ACL implementation only allows the collector to run the finalization when it knows that no peer has a reference, by keeping the Common Lisp reference to the object as a strong reference until the C ILU kernel informs the Common Lisp ILU runtime that no peer has a reference, in which case the Common Lisp reference is changed to a weak reference. In time this allows the collector to GC the object, and the finalization action is called. The action that needs to be taken is "null out" both the pointer from the CLOS object to the C object, via `(setf (ilu-cached-kernel-obj lisp-obj) nil)`, and "null out" the reference from the C object to the CLOS object, via `(register-language-specific-object (kernel-obj lisp-obj) 0)`. See '`ilu-franz.lisp`', `ilu::franz-shutdown-ilu-object`, for an example. The Franz ACL example also does these shutdowns in a separate thread, instead of doing them directly in the GC finalization process. This is because the shutdown actions may cause arbitrary callbacks into Common Lisp, some of which may not occur on the stack of the ACL scheduler, which may invoke the collector.

If you feel that it just isn't possible to hook your Common Lisp collector into the network GC, you can simply define `register-lisp-object` to ignore the *REFTYPE* parameter, and define `optional-finalization-hook` to expand to nothing. The result will be that no ILU object in your address space will ever be GC'ed, and that no true instance of a collectible ILU object type referenced by your process will ever be GC'ed anywhere in its true address space until your Common Lisp image disappears. This might also be a good starting point, just to get the other parts working.

C.3 Thread and/or Event Loops

Every address space into which ILU is loaded is implicitly a server. This is partially because ILU uses method calls internally, such as pinging garbage collection callbacks, and partially because it provides for recursive protocols, in which a "server" might call back to a "client" during the execution of a method call. This means that any implementation of ILU has to provide a way to execute incoming calls; which means that it has to provide a stack and thread of control in which to execute the "true" code of the method call. There are two mechanisms supported by ILU to associate a thread of control with an incoming request, threads and event loops. In the thread model, each request is executed in a thread associated with either the specific request (thread-per-request) or the connection on which the thread arrives (thread-per-client). In the event loop model, one thread of control is multiplexed between all uses by means of calls into particular "event handler" routines when some "event" is delivered to the process. Typical events are timer expirations, I/O available on file descriptors, UNIX signals. Other more application-specific events are possible, such as X Window System events or XView toolkit events.

For a threaded Common Lisp, the thread model is preferred. To support this, the implementor of the Common Lisp runtime must call the C procedure `ilu_SetWaitTech()` with two C-callable routines that provide ways to block the current thread until input or output is available on a particular file descriptor. He must call `ilu_SetMainLoop()` with a main loop struct that provides NULL procedures for the `ml_run`, `ml_exit`, `ml_register_input`, and `ml_register_output` fields, simple procedures that return `ilu_FALSE` for the `ml_unregister_input` and `ml_unregister_output` fields, and three C-callable procedures that implement creation, setting, and unsetting of alarms for the `ml_create_alarm`, `ml_set_alarm`, and `ml_unset_alarm` fields. Finally, he must provide C-callable procedures to describe his thread system's mutex and condition variable system to the ILU C kernel, and register them by calling `ilu_SetLockTech()`. See the Franz ACL implementation for an example of this. Note that the file '`ilu-process.lisp`' provides an implementation-independent veneer over various process systems. It would be useful to extend that, then use it in providing the specific thread mechanisms, rather than using your Common Lisp's threads directly.

For an non-threaded Common Lisp, the event loop model is available. In this, you divide up all computation in your application into event handlers, separate functions that are run when some event occurs, and initialize the system by calling some event handler dispatcher routine, often called the "main loop" of the system. ILU provides a default main loop in the kernel, which provides support for two kinds of events: timer expiration (ILU calls timers "alarms"), and input or output available on a UNIX file descriptor. This means that handler functions can be registered to be called when an event of one of these types occurs. The ILU event loop is also "recursive"; this means that event handlers can call back into the main loop to wait for something to occur. To use the ILU main loop, you must provide mainly a way to invoke the main loop, probably something like `ilu:xxx-main-loop`, where "xxx" is the name of your flavor of Common Lisp.

If the ILU main loop is for some reason not satisfactory, a Common Lisp-runtime-specific main loop can be substituted via a call to the ILU C kernel routine `ilu_SetMainLoop()`. This is often necessary to interoperate with UI toolkits like XView or Tk which believe that they own the main loop. Note that this main loop must provide all the functionality provided by the ILU main loop. A less-powerful main loop can be used *in addition to* the ILU main loop, by calling the ILU C kernel routine `ilu_AddRegisterersToDefault()`. See the comments in '*ILUSRC*`/runtime/kernel/iluxport.h`' for documentation of all of this.

In addition to making the appropriate calls into the ILU kernel to set up either threaded mode or event-loop mode, the Common Lisp runtime implementor must provide a few required function calls:

ilu::initialize-locking Function

This misnamed function is called by the generic ILU Common Lisp runtime to set up the interaction mode, start the scheduler if necessary, and in general do anything necessary to initialize the Common Lisp-flavor-specific Common Lisp runtime.

ilu::setup-new-connection-handler (*FN* `function`) (*SERVER* `C-pointer`) (*PORT* `C-pointer`) Function

This is called when a client connects to a kernel server, *SERVER*, implemented in this address space. It should arrange to apply *FN* to `(list` *SERVER PORT*`)` if a new incoming connection is received on *PORT*. *FN* should return `cl:nil` if no handler could be established, non-`cl:nil` otherwise. *SERVER* is the C address of an ILU kernel `ilu_Server`, *PORT* is the C address of an ILU kernel `ilu_Port`. The ILU C kernel routine `ilu_FileDescriptorOfMooringOfPort()` will return the UNIX file descriptor of the `ilu_Mooring` of an `ilu_Port`. In threaded Common Lisps, this will typically cause a thread to be forked, which will watch for connections to this port. In event-loop

Common Lisps, this will typically register *FN* as an event handler for "input available on the file descriptor of the mooring of *PORT*".

ilu::setup-connection-watcher (*FN* `function`) (*CONN* `C-pointer`) (*SERVER* `C-pointer`) Function

This is called when a new connection is setup. It should arrange things so that *FN* is applied to `(list` *CONN SERVER*`)` whenever input is available on *CONN*. *FN* should return non-`cl:nil` if the input was successfully handled, `cl:nil` otherwise. If *FN* ever returns `cl:nil`, the connection-watcher should be demolished. *CONN* is the C address of an ILU kernel `ilu_Connection`, and *SERVER* is the C address of an ILU kernel `ilu_Server`. The ILU C kernel routine `ilu_FileDescriptorOfConnection()` will return the UNIX file descriptor for an `ilu_Connection`. In threaded Common Lisps, this will typically fork a thread which will handle requests coming in on this connection. In event-loop Common Lisps, this will typically register *FN* as an event handler for "input available on the file descriptor of the connection".

C.4 Converting between character sets.

This section is not currently correct, but we are changing the Lisp runtime to make it correct.

ILU uses the ISO Latin-1 and Unicode (ISO 10646) character sets. Common Lisp uses a somewhat different version of 'character'. To provide for a mapping back and forth between ILU and Common Lisp, the runtime implementor must provide four macros:

ilu::construct-lisp-character-from-unicode (*UNICODE* `(unsigned-byte 16)`)) `=> character` Macro

ilu::determine-unicode-of-character (*LISP-CHAR* `character`) `=> Unicode-code` Macro

ilu::construct-lisp-character-from-latin-1 (*LATIN-1-CODE* `(unsigned-byte 8)`) `=> character` Macro

ilu::determine-latin-1-of-character (*LISP-CHAR* `character`) `=> ISO-Latin-1-code` Macro

which I trust are self-explanatory.

C.5 Support for Dynamic Object Creation

ILU allows the dynamic creation of objects. This means that a true module can create the true CLOS object for an ILU object in a lazy manner, when it is referenced. The mechanism for doing this is called *object tables*. An object table consists of 2 C-callable functions, one to create an object, given its instance handle, and one to free any storage associated with the object table. To support this mechanism, the Common Lisp port of ILU has to provide the following function:

ilu::create-object-table (*OBJECT-OF-IH-FN* `function`) (*FREE-SELF-FN* `function`) `=> C-pointer` Function

The function accepts two Lisp functions, and returns a pointer to a C struct of type `ilu_ObjectTable`, or the value `0`, if no object table pointer can be produced. The function will have to call into C space to actually produce the object table. Look at the Franz ACL implementation for an example of how to do this.

Appendix D Possible ISL Name Mappings for Target Languages

This note outlines a proposal for name mappings and restrictions; this proposal is not yet accepted. (Thanks to external standards such as CORBA, this proposal cannot be implemented for some languages, such as ANSI C.) **The mappings outlined here are not necessarily the ones used in the current ILU release.**

This proposal is about how to name things in the various programming languages, in a way that avoids name clashes. It imposes no restrictions on the ISL source. However, the mappings will be more straightforward if the ISL source avoids two things: (1) two or more concsecutive hyphens in a name, and (2) starting an interface or type name with “ilu-” (in any casing).

The first step in mapping an ISL to a programming language is to scan type and interface names for the substring “ilu-” (in any casing); wherever it occurs, we insert a trailing digit zero.

In a similar way, we next scan the name for sequences of hyphens. Wherever two or more hyphens appear consecutively, the digit zero (‘0’) is inserted after every other one, starting with inserting a zero after the second hyphen.

The following steps assume the first two steps have already been done.

Where tuples <N1, N2, ... Nk> of ISL names must be mapped into a flat programming namespace, we concatenate the ISL names, with a double hyphen (“–”) inserted between each.

Where ISL names (or tuples thereof) must be mapped, together with ILU-chosen names derived from the ISL names, into a flat programming namespace, the derived names begin with fixed strings specific to the derivation, where the fixed strings begin with “ilu-” (with any case), and a double hyphen is inserted between the fixed string and the ISL name.

Where ISL names (or tuples thereof), and possibly ILU-chosen names derived from the ISL names, must be mapped, together with a fixed set of ILU-chosen names, into a flat programming namespace, the fixed ILU-chosen names begin with “ilu-” (with any case) and do not include a double hyphen.

The final step is to translate hyphens to underscores, for programming languages that accept underscores but not hyphens in names.

Following is a specification of how names are mapped in each language. The notation "[N]" is used to denote the application of the first two steps and the last step. Examples of "[..]" are:

```
[Foo] => Foo
[foo-bar] => foo-bar
[wait----for---it-] => wait--0--0for--0-it-
[iluminate] => iluminate
[ilu---uli] => ilu-0--0uli
```

The mappings also use the notation "[[..]]" to denote the mapping of a type-reference.

D.1 C mapping

[This mapping, while clean, will never be adopted because of the more problematic mapping specified by the OMG's CORBA document.]

Item `N` from interface `I` is mapped to `[I]__[N]`. `[[I.N]] = [I]__[N]`; `[[N]] = [I]__[N]`, where `I` is the current interface.

An enumerated value named `V`, of type `T` in interface `I` is mapped to `[I]__[T]__[V]`.

A declaration of a record type `T` in interface `I` with fields `F1:TR1`, ... `Fn:TRn` is mapped to

```
typedef struct {[[TR1]] F1; ... [[TRn]] Fn} [I]__[T];
```

A declaration of a union type `T` in interface `I` of types `TR1`, ... `TRn` is mapped to

```
typedef enum {[[I.T]]__[[TR1]], ... [[I.T]]__[[TRn]]} ilu_tags__[[I.T]];
typedef struct {ilu_tags__[[I.T]] tag;
    union {
        [[TR1]] [[TR1]];
        ...
        [[TRn]] [[TRn]];
    } val;
} [[I.T]];
```

For passing exceptions through the method calls in interface `I`, the following auxiliary declaration is generated (supposing exceptions `ER1:TR1`, ... `ER2:TR2` are raised):

```
typedef struct {
    ilu_Exception returnCode;
    union {
        [[TR1]] [[ER1]];
        ...
        [[TRn]] [[ERn]];
    } val;
} ilu_Status__[I];
```

An object type named `T` in interface `I` with methods `M1`, ... `Mn` maps to

```
typedef ilu_Ojbect [[I.T]];
[result-type-1] [I]__[T]__[M1]([[I.T]] ilu_self,
    [[arg-type-1-1]] [arg-name-1-1], ...
    [[arg-type-1-k]] [arg-name-1-k]);
...
```

D.2 C++ mapping

Item `N` from interface `I` is mapped to `[I]__[N]`. `[[I.N]] = [I]__[N]`; `[[N]] = [I]__[N]`, where `I` is the current interface.

A declaration of an enumerated type named `T` in interface `I` containing values `V1`, ... `Vn` is mapped to `typedef enum {[V1], ... [Vn]} [I]__[T]`.

Record and union declarations are mapped as for C. The exception status declaration is as for C.

D.3 Modula-3 mapping

ILU interface `I` is mapped to Modula-3 interface `[I]`; within an interface, item `N` is mapped to item `[N]`. `[[I.N]] = [I].[N]`; `[[N]] = [N]`. For Modula-3, we also use the notation "((a type-reference))", defined by `((N)) = [N]` and `((I.N)) = [I]__[N]`.

A declaration of a record type `T` in interface `I` with fields `F1:TR1`, ... `Fn:TRn` is mapped to

```
TYPE [T] =  RECORD F1: [[TR1]]; ... Fn: [[TRn]]; END;
```

A declaration of a union type `T` in interface `I` of types `TR1`, ... `TRn` is mapped to

```
TYPE [T] = BRANDED OBJECT END;
TYPE [T]__((TR1)) = [T] BRANDED OBJECT v: [[TR1]] END;
...
TYPE [T]__((TRn)) = [T] BRANDED OBJECT v: [[TRn]] END;
```

A declaration of an enumerated type named `T` in interface `I` containing values `V1`, ... `Vn` is mapped to `TYPE [T] = {[V1], ... [Vn]};`

The type `Ilu.Object` has slots named `ilu_is_surrogate`, `Ilu_Get_Server`, `Ilu_Get_Type`, and `Ilu_Close_Surrogate`. Method names `M` are translated to `[M]`. For each object type `T`, the Modula-3 interface includes auxiliary procedures named `Ilu_Sbh_Import__[T]`, `Ilu_Name_Import__[T]`, and `Ilu_Get_Type__[T]`.

Index of Concepts

Index of Functions, Variables, and Types

T

W

This Page Intentionally Left "Blank".

Table of Contents